

Table of Contents

前言	1.1
java基础总结	1.2
JVM虚拟机	1.3
并发编程相关知识点	1.4
jdk内置JUC组件	1.4.1
JUC之AQS源码	1.4.1.1
JUC之ReentrantLock源码	1.4.1.2
JUC之Condition源码	1.4.1.3
guava框架	1.4.2
Disruptor框架	1.4.3
java8新特性	1.5
servlet总结	1.6
Spring总结	1.7
SpringMVC	1.7.1
NIO	1.8
算法总结	1.9
冒泡排序	1.9.1
堆排序	1.9.2
插入排序	1.9.3
归并排序	1.9.4
快速排序	1.9.5
选择排序	1.9.6
希尔排序	1.9.7
设计模式	1.10
单例模式	1.10.1
策略模式	1.10.2
分布式	1.11
分布式事务	1.11.1
Zookeeper	1.11.2
redis	1.11.3
LINUX+DOCKER	1.12
附录	1.13
idea快捷键大全	1.13.1

前言

java基础总结

1.1 对象的概念

- 万物皆对象
- 程序是一组对象，通过信息传递告知彼此该做什么。
- 每个对象都有自己的存储空间，可容纳其他对象。
- 每个对象都有一个类型。每个对象都是某个类的实例。
- 同一类的对象都能接口相同的类型。

1.3 抽象类

- 抽象方法：只有声明，没有具体的实现。
- 如果有一个类含有抽象方法，则这个类抽象类，需由abstract修饰。特点：
- 抽象方法必须是public或者protected
- 抽象类不能实例化
- 如果子类不是不是抽象类，则必须实现父类的抽象方法

1.3 接口

- 接口中定义的变量被隐式指定为public static final。
- 接口中的所有方法不能有具体的实现。在java8中，可用default关键字在接口中实现默认方法。

1.4 接口与抽象类的区别

- 抽象类中的成员变量可以是各种类型的，而接口中的变量是public static final
- 接口不能有静态代码块以及静态方法，而抽象方法可以
- 一个类只能实现一个抽象类，但是一个类可以实现多个接口。
- 抽象类是对事物的抽象，接口是对行为的抽象

1.5 重写equal()方法的注意点

- 自反性：对于任何非空引用，x.equals(x)应该返回true
- 对称性：对于任何引用x和y，如果x.equals(y)返回为true，那么y.equals(x)也应该返回true
- 传递性：对于任何引用，x，y和z，如果x.equals(y)返回为true，那么y.equals(x)也应该返回true
- 一致性：如果x和y引用的对象没有发生变化，返回调用x.equals(y)应该返回相同的结果
- 对于任何非空引用x，x.equals(null)应该返回false

1.6 对象序列化

对象序列化是以特殊的文件格式存储对象数据的。

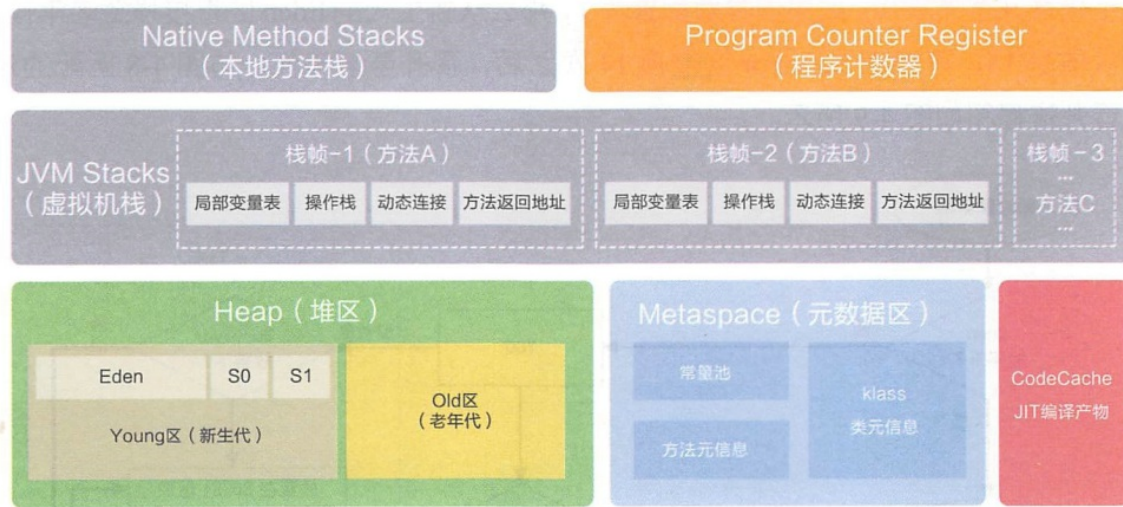
1.7 类的加载

Java默认提供的三个ClassLoader：

- Bootstrap ClassLoader：称为启动类加载器，是Java类加载层次中最顶层的类加载器，负责加载JDK中的核心类库
- Extension ClassLoader：称为扩展类加载器，负责加载Java的扩展类库，默认加载JAVA_HOME/jre/lib/ext/目下的所有jar。
- App ClassLoader：称为系统类加载器，负责加载应用程序classpath目录下的所有jar和class文件。

ClassLoader使用的是双亲委托模型来搜索类的（避免重复加载），从上至下搜索。每个ClassLoader实例都有一个父类加载器的引用。JVM在判定两个class是否相同时，不仅要判断两个类名是否相同，而且要判断是否由同一个类加载器实例加载的。

JVM虚拟机



1.1 程序计数器（线程私有）

当前线程所执行的字节码的行号指示器，每条线程都要有一个独立的程序计数器。（指令的执行都是通过抢占cpu资源的（cpu分配时间片），每个线程可能执行了一段时间后，就被其他线程抢占了资源。所以需要程序计数器来记录当前执行到的行号。）

对于java方法，计数器记录的是虚拟机字节码指令的地址。对于Native方法，则为空。

1.2 虚拟机栈（线程私有）

是描述java方法执行的内存模型。每个方法在运行的同时都会创建一个栈帧，用于存储局部变量表，操作数栈，动态链接，方法出口等信息。每一个方法从调用直至执行完成的过程，就对应一个栈帧在虚拟机栈中入栈到出栈的过程。

栈帧随着方法调用而创建，随着方法结束而销毁。

1.3 本地方法区（线程私有）

类似虚拟机栈。区别在于本地方法栈为Native方法服务。

1.4 堆（运行时数据区，线程共享）

创建的对象和数组都保存在java堆内存中，也是垃圾收集器进行垃圾收集最重要的内存区域。

1.5 方法区/永久代（线程共享）

用于存储JVM加载的类信息，

1.6 垃圾回收算法

- 标记-清除算法：该算法从每个GC Roots出发，依次标记有引用关系的对象，最后将没有被标记的对象清除。此种算法会带来大量的空间碎片，导致需要一个较大连续空间时容易触发FullGC。
- 标记-整理算法：首先从GC Roots出发标记存活的对象，然后将存活对象整理到内存空间的一端，形成连续的已使用空间，最后把已使用空间之外的部分全部清理掉，这样就不会出现空间碎片的问题。
- "Mark-Copy"算法：为了能够并行的标记和整理将空间分为两块，每次只激活其中一块，垃圾回收时只需要把存活的对象复制到另一块未激活空间上，将未激活空间标记为已激活，将已激活空间标记为未激活，然后清除原空间中的原对象。为主流的YGC算法进行新生代的垃圾回收。

1.7 垃圾回收器

- serial回收器：主要应用于YGC的垃圾回收器，采用串行单线程的方式完成GC任务。在垃圾回收的某个阶段会暂停整个应用程序的执行。
- CMS回收器：回收停顿时间较短，目前比较常用的垃圾回收器。他通过初始标记、并发标记、重新标记、并发清除四个步骤完成垃圾回收工作。由于CMS采用的"标记-清除算法"，因此产生大量的空间碎片。为了解决这个问题，CMS可以通过配置-XX:+UseCMSCompactAtFullCollection参数，强制JVM在FGC完成后对老年代进行压缩，执行一次空间碎片整理。为了减少STW次数，CMS也可以通过配置-XX:+UseCMSCompactAtFullCollection=n，即在执行n次FGC后，JVM再在老年代执行空间碎片整理。
- G1：-XX:+UseG1GC。与CMS相比，G1具备压缩功能，能避免碎片问题。G1将Java堆空间分割成了若干相同大小的区域，即region，包括Eden、Survivor、Old、Humongous四种类型。其 中，Humongous是特殊的Old类型，专门放置大型对象。这样的划分方式意味着不需要一个连续的内存空间管理对象。G1将空间分为多个区域，优先回收垃圾最多的区域。G1采用的是"Mark-copy"，有非常好的空间整合能力，不会产生大量的空间碎片。G1的一大优势在于可预测的停顿时间，能够尽可能快地在指定时间内完成垃圾回收任务。在JDK11中，已经将G1设为默认垃圾回收器，通过jstat命令可以查看垃圾回收情况。

并发编程相关知识点

1.1 线程的状态

1. **New**（新创建）：新创建一个新的线程对象对象后，在调用他的**start**（）方法，系统会为此线程分配CPU资源，使其处于**Runnable**（可运行状态），这是一个准备运行的阶段。如果线程抢占到CPU资源，此线程就处于**Running**（运行）状态。
2. **Runnable**（可运行） **Runnable**状态和**Running**状态可相互切换，因为有可能线程运行一段时间后，有其他高优先级的线程抢占了CPU资源，这时**Running**状态变成了**Runnable**状态。线程进入**Runnable**状态大体分为如下几种情况：
 - 调用**sleep**（）方法后经过的时间超过了指定的休眠时间
 - 线程调用的堵塞IO已经返回，堵塞方法执行完毕
 - 线程成功地获取了试图同步的监视器
 - 线程处于等待某个通知，其他线程发出了通知。
 - 处于挂起状态的线程调用了**resume**回复方法。
3. **Blocked**（被堵塞）例如遇到一个IO操作，此时CPU处于空闲状态，可能会转而把CPU时间片分配给其他的线程，这是也可以成为“暂停”状态。
 - 线程调用**sleep**（）方法，主动放弃占用的处理器资源。
 - 线程调用了堵塞式IO方法，在该方法返回前，该线程堵塞
 - 线程试图获得一个同步监视器，但该同步监视器正被其他线程所持有。
 - 线程等待某个通知。
 - 程序调用了**suspend**方法将该线程挂起。此方法容易导致死锁，尽量避免使用该方法。
4. **run**（）：运行结束进入销毁阶段，整个线程执行完毕。
5. **join**（）：使所属线程对象x正常执行**run**（）方法中的任务，而使当前线程z无限期堵塞，等待线程x销毁后再继续执行线程z后面的代码。**join**具有使线程排队运行的作用，有些类似同步的运行效果。**join**与**synchronized**的区别是：**join**在内部使用**wait**方法进行等待，而**synchronized**使用的“对象监视器”原理做同步。
6. **Waiting**（等待）
7. **Timed**（waiting）
8. **Terminated**（被终止）

1.2 指令重排

指令重排是指在程序执行过程中, 为了性能考虑, 编译器和CPU可能会对指令重新排序。

不能重排的指令

- 程序顺序原则，一个线程内保证语义的串行性。
- **volatile**规则：**volatile**变量的写先于读发生，这保证了**volatile**变量的可见性。
- 锁规则：解锁（**unlock**）必然发生在随后加锁（**lock**）前。
- 传递性：**A**先于**B**，**B**先于**A**，那么**A**必然先于**B**。
- 线程的**start**（）方法先于它的每个动作。
- 线程的所有操作先于线程的终结（**Thread.join**）
- 线程的中断（**interrupt**（））先于被中断线程的代码。
- 对象的构造函数的执行，结束先于**finalize**（）方法。

1.3 CAS算法

cas算法过程是：它包含三个参数**CAS**（**V,E,N**），其中**V**表示呀更新的变量，**E**表示预期值，**N**表示新值。仅当**V**值等于**E**值时，才会将**V**的值设为**N**，如果**V**值和**E**值不同，说明已经有其他线程做了更新，则当前线程什么都不做。最后，**CAS**返回当前**V**的真实值。**CAS**操作是抱着乐观的态度进行的，它总认为自己可以成功完成操作。当多个线程同时使用**CAS**操作一个变量时，只有一个会胜出，并成功更新，其余的均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。

1.4 死锁的概念

死锁的概念 死锁就是两个或者多个线程互相占用对方需要的资源，而都不进行释放，导致彼此之间相互等待对方释放资源，产生了无限制等待的现象。死锁一旦发生，如果没有外力介入，这种等待将永远存在，从而对程序产生严重的影响。

jdk内置JUC组件

1.1 JUC核心组件AQS

AQS是类AbstractQueuedSynchronizer的简称（以下均以AQS代替），提供了一种实现堵塞锁（独占锁）和一系列FIFO等待队列（共享锁）的同步框架。独占锁：此代码有且只有一个线程能够执行，如ReentrantLock 共享锁：多个线程可同时获取锁，如Semaphore/CountDownLatch

1.2 Lock相关概念

Lock作用:

- 锁用来保护代码片段，任何时刻只能有一个线程执行被保护的代码。
- 锁可以管理试图进入被保护代码段的线程。
- 锁可以拥有一个或多个相关的条件对象。
- 每个条件对象管理那些已经进入被保护的代码段但还不能运行的线程。

公平与非公平锁: 公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的FIFO先进先出顺序。而非公平锁是一种获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁，这种方式肯呢造成某些线程一直拿不到锁，结果也就是不公平的了。

1.2.1 ReentrantLock (默认非公平锁)

重入锁（ReentrantLock）是一种递归无阻塞的同步机制。它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续） synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

常用方法:

- lock():
 - 如果该锁没有被另一个线程保持，则获取该锁并立即返回，将锁的保持计数设置为 1。
 - 如果当前线程已经保持该锁，则将保持计数加 1，并且该方法立即返回。
 - 如果该锁被另一个线程保持，则出于线程调度的目的，禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态，此时锁保持计数被设置为 1
- lockInterruptibly()
 - 如果当前线程未被中断，则获取锁。
 - 如果该锁没有被另一个线程保持，则获取该锁并立即返回，将锁的保持计数设置为 1。
 - 如果当前线程已经保持此锁，则将保持计数加 1，并且该方法立即返回。
 - 如果锁被另一个线程保持，则出于线程调度目的，禁用当前线程，并且在发生以下两种情况之一以前，该线程将一直处于休眠状态(锁由当前线程获得；或者 其他某个线程中断当前线程。)
 - 如果当前线程获得该锁，则将锁保持计数设置为 1。
 - 此方法是一个显式中断点，所以要优先考虑响应中断，而不是响应锁的普通获取或重入获取。
- tryLock():仅在调用时锁未被另一个线程保持的情况下，才获取该锁。
 - 如果该锁没有被另一个线程保持，并且立即返回 true 值，则将锁的保持计数设置为 1。即使已将此锁设置为使用公平排序策略，但是调用 tryLock() 仍将 立即获取锁（如果有可用的），而不管其他线程当前是否正在等待该锁。在某些情况下，此“闯入”行为可能很有用，即使它会打破公平性也如此。如果希望遵守此锁的公平设置，则使用 tryLock(0, TimeUnit.SECONDS)，它几乎是等价的（也检测中断）。
 - 如果当前线程已经保持此锁，则将保持计数加 1，该方法将返回 true。
 - 如果锁被另一个线程保持，则此方法将立即返回 false 值。

1.2.2 Conditon

Condition主要是用来处理线程之间的通信，当满足某一条件时，唤醒其他的线程。主要配合重入锁完成等待唤醒的操作。

1.3 synchroized

相关定义:

- 调用关键字synchronized生命的方法一定是排队运行的
- 避免数据出现交叉的情况，使用synchronized关键词进行同步
- 关键词synchronized拥有锁重入的功能，也就是在使用synchronized时，当一个线程得到一个对象锁后，在此请求此对象锁时是可以再次得到该对象锁的。
- 当一个线程执行的代码出现异常时，其所持有的锁会自动释放。
- 同步不可以被继承。
- synchronized同步块 锁非this对象相比synchronized（this）更加灵活，当一个方法中有多个同步块时，不用竞争this对象锁。

关于synchronized（非this对象x）的写法是将x对象本身作为"对象监视器"，有如下三个结论：

- 当多个线程同时执行synchronized（X）{}同步代码块时呈同步效果。
- 当其他线程执行x对象中synchronized同步方法时呈同步效果。
- 当其他线程执行x对象方法中的synchronized（this）代码块时也呈现同步效果。
- 如果其他线程调用不加synchronized关键字的方法时，还是异步调用。
- synchronized关键字加到非static静态方法是给对象上锁，而教导static方法上则是对Class类加锁。
- 对于String对象，不要用作对象锁。String a="A",String b="A", a=b，导致线程会使用同一个对象锁
- 在设计程序时，要避免双方互相持有对方锁的情况，会导致死锁。

1.4 volatile

volatile只保证可见性，不保证原子性。

使用场景:

1. 写入变量时并不依赖变量的当前值。或者能够确定保证只有单一的线程能修改变量的值。
2. 变量不需要与其他状态变量共同参与不变约束。

3. 访问变量的时候，没有其他原因需要加锁。

volatile和**synchronized**比较

1. 关键字**volatile**是线程同步的轻量实现，所以**volatile**性能肯定比**synchronized**好，并且**volatile**只能修饰于变量，而**synchronized**可以修饰方法以及代码块。
2. 多线程访问**volatile**不会发生堵塞，而**synchronized**会出现堵塞。
3. **volatile**只能保证数据的可见性，但不能保证原子性；而**synchronized**可以保证原子性，也可间接保证可见性，因为它会将私有内存和公共内存中的数据做同步
4. **volatile**解决的是变量在多个线程之间的可见性，而**synchronized**解决的是多个线程之间访问资源的同步性。

1.5 ThreadLocal

ThreadLocal提供了**get**和**set**的访问器，为每个使用它的线程维护一份单独的拷贝。所以**get**返回的都是当前线程设置的最新值。

ThreadLocal在内部维护了一个**ThreadMap**用来映射线程的独有变量。

1.6 Future+callable

所谓异步调用其实就是实现一个可无需等待被调用函数的返回值而让操作继续运行的方法。在 **Java** 语言中，简单的讲就是另启一个线程来完成调用中的部分计算，使调用继续运行或返回，而不需要等待计算结果。但调用者仍需要取线程的计算结果。

CompletableFuture:

- **supplyAsync**: 执行一个异步请求，并返回一个**future**
- **thenApply**: 对结果应用一个函数
- **thenCompose**: 对结果调用函数并执行返回的**future**
- **handle**: 处理结果或错误
- **thenAccept**: 类似于 **thenApply**, 不过结果为 **void**
- **whenComplete**: 类似于 **handle**, 不过结果为 **void**
- **thenRun**: 执行 **Runnable**, 结果为 **void**
- **thenCombine**: 执行两个动作并用给定函数组合结果
- **thenAcceptBoth**: 与 **thenCombine** 类似， 不过结果为 **void**
- **runAfterBoth**: 两个都完成后执行 **_able**
- **applyToEither**: 得到其中一个的结果时， 传入给定的函数
- **acceptEither**: 与 **applyToEither** 类似， 不过结果为 **void**
- **runAfterEither**: 其中一个完成后执行 **runnable**
- **static allOf**: 所有给定的 **future** 都完成后完成， 结果为 **void**
- **static anyOf**: 任意给定的 **future** 完成后则完成， 结果为 **void**

1.7 SynchronousQueue（同步队列）

同步队列是一种将生产者与消费者线程配对的机制。当一个线程调用 **SynchronousQueue** 的 **put** 方法时，它会阻塞直到另一个线程调用 **take** 方法为止，反之亦然。与 **Exchanger** 的情况不同，数据仅仅沿一个方向传递，从生产者到消费者。即使 **SynchronousQueue** 类实现了 **BlockingQueue** 接口，概念上讲，它依然不是一个队列。它没有包含任何元素，它的 **size** 方法总是返回 0。

1.8 堵塞队列

用处：对于许多线程问题，可以通过使用一个或多个队列以优雅且安全的方式将其形式化。生产者线程向队列插入元素，消费者线程则取出它们。使用队列，可以安全地从一个线程向另一个线程传递数据。

常用堵塞队列

- **ArrayBlockingQueue**: 构造一个带有指定容量和公平性设置的堵塞队列。该队列用循环数组实现。
- **LinkedBlockingQueue/LinkedBlockingDeque**: 构造一个无上限的堵塞队列或双向队列，用链表实现。
- **DelayQueue**: 构造一个包含 **Delayed** 元素的无界的阻塞时间有限的阻塞队列。只有那些延迟已经超过时间的元素可以从队列中移出。
- **PriorityBlockingQueue**: 构造一个无边界阻塞优先队列，用堆实现。
- **ConcurrentLinkedQueue**: 构造一个可以被多线程安全访问的无边界非阻塞的队列。
- **ConcurrentSkipListSet**: 构造一个可以被多线程安全访问的有序集。第一个构造器要求元素实现 **Comparable** 接口。

1.9 CountdownLatch闭锁

- 确保一个计算不被执行，直到它需要的资源初始化。
- 确保一个服务不会开始，直到它依赖的其他服务都已经开始。
- 等待，直到活动的所有部分都为继续处理做好充分准备。**CountDownLatch**是一个灵活的闭锁实现。允许一个或多个线程等待一个事件集的发生。闭锁状态包括一个计数器，初始化为一个整数。用来表现需要等待的事件数。**countDown**方法针对计数器做减操作，表示一个事件已经发生了，而**await**方法等待计数器达到零，此时所有需要等待的事件都已经发生。如果计数器入口时值为非零，**await**会一直堵塞直到计数器为零，或者等待线程中断以及超时。

1.10 CyclicBarrier同步屏障

类似于闭锁。与闭锁不同之处在于，所有的线程必须同时到达关卡点，才能继续处理。闭锁等待的是事件；而同步屏障等待的是其他的线程。常用示例比如：可将一个任务分割成多个子部分，然后再整合。

1.11 Semaphore计数信号量

- 用来控制能够同时访问某特定资源的活动的数量。
- 计数信号量可以用来实现资源池或者给一个容器限定边界。
- 一个**Semaphore**管理一个有效的许可，许可的除量通过构造函数传递给**semaphore**活动能够获得许可（只要还有剩余许可），并在使用之后释放许可，如果没有可用的许可，则**acquire**会被堵塞，直到有可用的为止。
- 常见的信号量使用即数据库连接池。

1.12 Exchanger交换器

当两个线程在同一个数据缓冲区的两个实例上工作的时候， 就可以使用交换器 (Exchanger) 典型的情况是， 一个线程向缓冲区填入数据， 另一个线程消耗这些数据。当它们都完成以后，相互交换缓冲区。

1.13 线程池相关参数

核心ThreadPoolExecutor参数：

- **corePoolSize**: 指定线程池中的线程数量
- **maximumPoolSize**: 指定了线程池中的最大线程数量。
- **keepAliveTime**: 当线程池线程数量超过**corePoolSize**，多余的空闲线程的存活时间，即超过**corePoolSize**的空闲线程，在多长时间内会被销毁。
- **unit**: **keepAliveTime**的单位
- **workQueue**: 任务队列，被提交但未被执行的任务。
- **threadFactory**: 线程工厂，用于创建线程，一般用默认的既可
- **handler**: 拒绝策略。当任务太多来不及处理，如何拒绝任务。

workQueue说明: 是一个BlockingQueue接口对象。仅用于存放Runnable对象。

- **直接提交队列**: 该功能由SynchronousQueue对象提供。SynchronousQueue没有容量，每一个插入操作都要等待一个相应的删除操作，同理删除。如果使用SynchronousQueue，则提交的任务不会被真实的保存，而总是将新任务提交给线程执行，如果没有空闲的线程，则尝试创建新的线程，如果线程已经达到最大值，则执行拒绝策略。
- **有界的任务队列**: ArrayBlockingQueue实现。当有新的任务需要执行，如果线程池的实际线程数小于**corePoolSize**，则会优先创建新的线程，若大于**corePoolSize**，则会将新任务加入等待队列，若等待队列已满，无法加入，则在总线程不大于**maximumPoolSize**的前提下，创建新的线程执行任务。若大于**maximumPoolSize**，则执行拒绝策略。
- **无界的任务队列**: LinkBlockingQueue类实现。与有界队列相比，除非系统资源耗尽，否则无界队列不存在任务入队失败的情况，当有新的任务到来，系统的线程数小于**corePoolSize**，线程池会生成新的线程执行任务，当系统的线程数达到**corePoolSize**，就不会继续增加了。若后续仍有新的任务加入，而有没有空闲的线程资源，则任务直接进入队列等待。若任务创建和处理的速度差异很大，无界队列会保持快速增长，直到耗尽系统内存。
- **优先任务队列**: PriorityBlockingQueue实现。可以控制任务执行的先后顺序。它是一个特殊的无界队列。

拒绝策略说明: (可扩展RejectedExecutionHandler接口)

- **AbortPolicy**: 该策略会直接抛出异常，阻止系统工作
- **CallRunsPolicy**: 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能急剧下降。
- **DiscardOldestPolicy**: 该策略将丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。
- **DiscardPolicy**: 该策略默默丢弃无法处理的任务，不予任何处理。

扩展: ThreadPoolExecutor提供了beforeExecute(),afterExecute和terminated()三个方法用来对线程池进行控制

JUC核心之AQS

Node节点（线程状态维护）

AQS中维护线程状态是通过一个内部类Node来维护的。源码如下

```
static final class Node {
    //节点在共享模式下的标记
    static final Node SHARED = new Node();
    //节点在独占锁模式下的标记
    static final Node EXCLUSIVE = null;
    //标志着线程被取消
    static final int CANCELLED = 1;
    //标志着后继线程(即队列链表的下一个节点)需要被阻塞。
    static final int SIGNAL = -1;
    //标志着线程在Condition条件上等待阻塞
    static final int CONDITION = -2;
    //标志着下一个acquireShared方法线程应该被允许，即锁可向下一级传播。(用于共享锁)
    static final int PROPAGATE = -3;
    //维护锁的状态
    volatile int waitStatus;

    //前驱节点
    volatile Node prev;
    //后继节点
    volatile Node next;
    //获取同步状态的线程,进入此节点队列的线程
    volatile Thread thread;
    //等待节点的后继节点。如果当前节点是共享的，那么这个字段是一个SHARED常量，也就是说节点类型（独占和共享）和等待队列中的后继节点共用一个字段。
    Node nextWaiter;

    final boolean isShared() {
        return nextWaiter == SHARED;
    }

    //返回前驱节点
    final Node predecessor() throws NullPointerException {
        Node p = prev;
        if (p == null)
            throw new NullPointerException();
        else
            return p;
    }

    Node() {
    }
    //Used by addWaiter
    Node(Thread thread, Node mode) {
        this.nextWaiter = mode;
        this.thread = thread;
    }
    // Used by Condition
    Node(Thread thread, int waitStatus) {
        this.waitStatus = waitStatus;
        this.thread = thread;
    }
}
```

独占锁

四维导图



https://blog.csdn.net/qq_38661798

1. 获取锁

```
public final void acquire(int arg) {
```

```

//尝试获取锁资源，失败则创建等待队列，此处的tryAcquire由子类实现
if (!tryAcquire(arg) &&
    acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
    //中断当前线程
    selfInterrupt();
}

```

1. 将Node加入等待队列

```

private Node addWaiter(Node node) {
    Node node = new Node(Thread.currentThread(), mode);
    //记录当前尾节点
    Node pred = tail;
    //此处为了提高性能，直接进入入队操作，失败则继续处理（多线程下竞争）
    if (pred != null) {
        //设置前驱节点
        node.prev = pred;
        // CAS操作，多个线程同时进行入队，只有一根线程能入队成功，其余的继续执行
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    //上一步失败的节点入队
    enq(node);
    return node;
}

```

1. 节点加入队尾

```

private Node enq(final Node node) {
    //自旋锁，失败重试
    for (;;) {
        //记录队尾元素
        Node t = tail;
        //初始化队列
        if (t == null) {
            //CAS操作，只有一个线程可以初始化头结点成功，其余的都要重复执行循环体
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            //入队的节点指向队尾，此处会有多个节点指向队尾
            node.prev = t;
            //CAS操作，保证多线程下只有一个节点真正指向队尾节点，其余失败重试
            if (compareAndSetTail(t, node)) {
                t.next = node;
                //该循环体唯一退出的操作，就是入队成功（否则就要无限重试）
                return t;
            }
        }
    }
}

```

1. 将节点加入等待队列后，先判断是否能够获取锁的资源，不能则进入挂起状态等待唤醒

```

final boolean acquireQueued(final Node node, int arg) {
    //锁资源获取失败标记位
    boolean failed = true;
    try {
        //线程被中断标识
        boolean interrupted = false;
        // 自旋锁等待被唤醒
        for (;;) {
            //获取当前节点的前置节点
            final Node p = node.predecessor();
            //如果前置节点就是头结点，则尝试获取锁资源
            if (p == head && tryAcquire(arg)) {
                //将当前节点设置为头结点
                setHead(node);
                p.next = null; //帮助GC
                //表示锁资源成功获取
                failed = false;
                //返回中断标记，表示当前节点是被正常唤醒还是被中断唤醒
                return interrupted;
            }
            //如果没有获取锁成功，则进入挂起逻辑
            //无论被中断或者正常唤醒，都会进行重新获取锁。成功则释放，失败则挂起
            // acquireInterruptibly不同在于，线程堵塞时被中断会抛出异常
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        //获取锁失败处理逻辑
        if (failed)
            cancelAcquire(node);
    }
}

```

1. 检测节点状态，判断是否可进入挂起状态，同时剔除前边已经取消的节点

```

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    //获取前置节点的waitStatus
    int ws = pred.waitStatus;
}

```

```

//如果处于等待唤醒状态，则返回为true进行堵塞
if (ws == Node.SIGNAL)
    return true;
if (ws > 0) {
    //循环查找没有被取消的前节点，相当于把之前取消的节点从队列中剔除出去
    do {
        node.prev = pred = pred.prev;
    } while (pred.waitStatus > 0);
    pred.next = node;
} else {
    //将前置节点状态设置Node.SIGNAL，执行上一步的自旋，进行堵塞
    compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
}
return false;
}

```

2. 堵塞线程

```

private final boolean parkAndCheckInterrupt() {
    //堵塞线程
    LockSupport.park(this);
    //被唤醒之后，返回中断标记，即如果是正常唤醒则返回false，如果是由于中断醒来，就返回true
    return Thread.interrupted();
}

```

3. 获取锁或则加入等待队列失败后的处理

```

private void cancelAcquire(Node node) {
    if (node == null)
        return;

    node.thread = null;

    Node pred = node.prev;
    // 跳过所有已经取消的前置节点，跟上面的那段跳转逻辑类似
    while (pred.waitStatus > 0)
        node.prev = pred = pred.prev;

    //取得的未取消节点的后一节点
    Node predNext = pred.next;
    //把当前节点waitStatus置为取消，这样别的节点在处理时就会跳过该节点
    node.waitStatus = Node.CANCELLED;

    //如果当前是尾节点，则直接删除，即出队
    if (node == tail && compareAndSetTail(node, pred)) {
        compareAndSetNext(pred, predNext, null);
    } else {
        int ws;
        if (pred != head &&
            ((ws = pred.waitStatus) == Node.SIGNAL ||
             (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&
            pred.thread != null) {
            Node next = node.next;
            //如果当前节点的前置节点不是头节点且它后面的节点等它唤醒（waitStatus小于0），如果当前节点的后继节点没有被取消就把前置节点跟后置节点进行连接
            if (next != null && next.waitStatus <= 0)
                compareAndSetNext(pred, predNext, next);
        } else {
            unparkSuccessor(node);
        }
    }

    node.next = node; // help GC
}
}

```

1. 锁的释放

```

public final boolean release(int arg) {
    // 调用tryRelease方法，尝试去释放锁，由子类具体实现
    if (tryRelease(arg)) {
        Node h = head;
        //如果队列头节点的状态不是0，那么队列中就可能存在需要唤醒的等待节点。
        //
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}

```

1. 线程唤醒

```

private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    //如果当前节点的状态小于0，则将该节点置为初始状态，标识节点已完成
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    Node s = node.next;
    // 如果下一个节点为null，或者状态是已取消，那么就要寻找下一个非取消状态的节点
    if (s == null || s.waitStatus > 0) {
        s = null;
        // 从队列尾向前遍历，直到遍历到node节点
        for (Node t = tail; t != null && t != node; t = t.prev)

```

```

        if (t.waitStatus <= 0)
            s = t;
    }
    if (s != null)
        //唤醒线程
        LockSupport.unpark(s.thread);
}

```

共享锁的实现

1. 释放共享锁

```

private void doAcquireShared(int arg) {
    // 创建一个共享锁节点
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            // 获取下一节点
            final Node p = node.predecessor();
            // 前节点为头结点
            if (p == head) {
                // 尝试释放共享锁
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    // 获取锁后的唤醒操作
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    // 如果是被中断唤醒，则设置中断标记为
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }
            // 挂起逻辑同独占锁
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            // 失败逻辑同独占锁
            cancelAcquire(node);
    }
}

```

1. 线程唤醒

```

private void setHeadAndPropagate(Node node, int propagate) {
    // 记录头结点以便检查
    Node h = head;
    // 设置当前节点为新的头节点
    setHead(node);
    // 节点满足如下条件则需要进行唤醒线程
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        // 如果节点s是空或者共享模式节点，那么就要唤醒共享锁上等待的线程
        if (s == null || s.isShared())
            doReleaseShared();
    }
}

```

2. 具体的唤醒逻辑

```

private void doReleaseShared() {
    for (;;) {
        Node h = head;
        // 从头结点开始唤醒线程
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            // 后继节点需要被唤醒
            if (ws == Node.SIGNAL) {
                // 将点h的状态设置成0，如果设置失败，就继续循环，再试一次
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue; // loop to recheck cases
                // 执行唤醒操作
                unparkSuccessor(h);
            }
            // 如果后继节点不需要被唤醒，则设置当前节点为PROPAGATE确保以后可以传递
            else if (ws == 0 &&
                !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue; // loop on failed CAS
        }
        // 头结点没有发生变化，则表示设置完成，退出循环
        if (h == head)
            break;
    }
}

```

```

//唤醒node节点的后继节点
private void unparkSuccessor(Node node) {
    //节点的等待状态
    int ws = node.waitStatus;
    //置零当前线程所在的结点状态，允许失败
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    //后继节点
    Node s = node.next;
    //后继结点为空或者状态大于0，则无需处理
    if (s == null || s.waitStatus > 0) {
        s = null;
        //从尾节点遍历，找到下一个可唤醒的节点
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        //唤醒线程
        LockSupport.unpark(s.thread);
}## 定义

```

AQS是类AbstractQueuedSynchronizer的简称（以下均以AQS代替），提供了一种实现堵塞锁（独占锁）和一系列FIFO等待队列（共享锁）的同步框架。

独占锁：此代码有且只有一个线程能够执行，如ReentrantLock

共享锁：多个线程可同时获取锁，如Semaphore/CountDownLatch

AQS源码

Node节点（线程状态维护）

AQS中维护线程状态是通过一个内部类Node来维护的。源码如下

static final class Node { //节点在共享模式下的标记 static final Node SHARED = new Node(); //节点在独占锁模式下的标记 static final Node EXCLUSIVE = null; //标志着线程被取消 static final int CANCELLED = 1; //标志着后继线程(即队列链表的下一个节点)需要被阻塞. static final int SIGNAL = -1; //标志着线程在Condition条件上等待阻塞 static final int CONDITION = -2; //标志着下一个acquireShared方法线程应该被允许，即锁可向下一级传播。(用于共享锁) static final int PROPAGATE = -3; //维护锁的状态 volatile int waitStatus;

```

//前驱节点
volatile Node prev;
//后继结点
volatile Node next;
//获取同步状态的线程,进入此节点队列的线程
volatile Thread thread;
//等待节点的后继节点。如果当前节点是共享的，那么这个字段是一个SHARED常量，也就是说节点类型（独占和共享）和等待队列中的后继节点共用一个字段。
Node nextWaiter;

final boolean isShared() {
    return nextWaiter == SHARED;
}

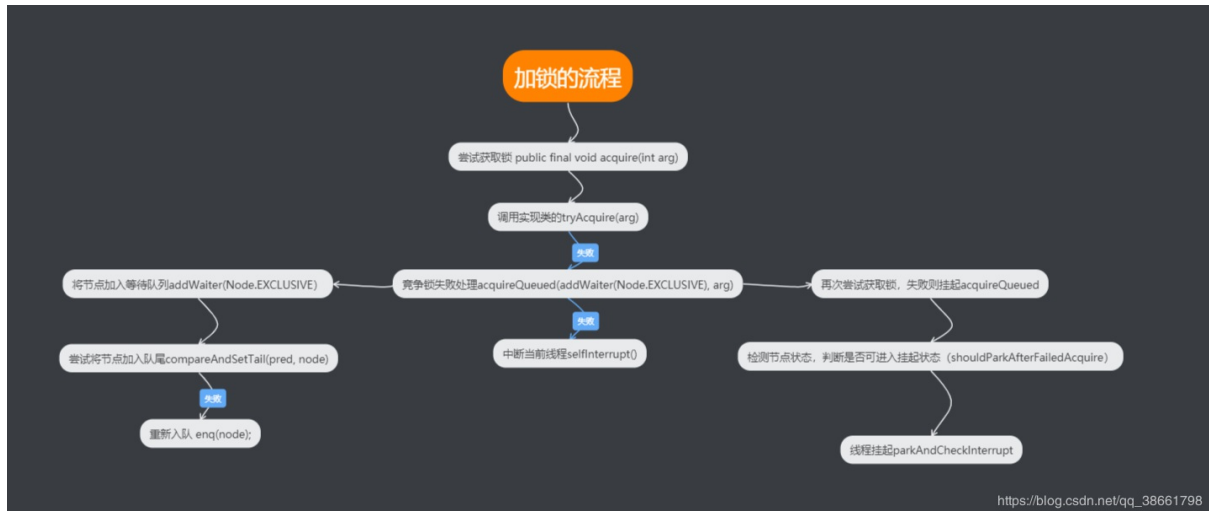
//返回前驱节点
final Node predecessor() throws NullPointerException {
    Node p = prev;
    if (p == null)
        throw new NullPointerException();
    else
        return p;
}

Node() {
}
//Used by addWaiter
Node(Thread thread, Node mode) {
    this.nextWaiter = mode;
    this.thread = thread;
}
// Used by Condition
Node(Thread thread, int waitStatus) {
    this.waitStatus = waitStatus;
    this.thread = thread;
}
}

```

...

独占锁


https://blog.csdn.net/qq_38661798

1. 获取锁

```
public final void acquire(int arg) { //尝试获取锁资源, 失败则创建等待队列, 此处的tryAcquire由子类进行实现 if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) //中断当前线程 selfInterrupt(); }
```

1. 将Node加入等待队列

```
private Node addWaiter(Node mode) { Node node = new Node(Thread.currentThread(), mode); //记录当前尾节点 Node pred = tail; //此处为了提高性能, 直接进行入队操作, 失败则继续处理 (多线程下竞争) if (pred != null) { //设置前驱节点 node.prev = pred; // CAS操作, 多个线程同时进行入队, 只有一根线程能入队成功, 其余的继续执行 if (compareAndSetTail(pred, node)) { pred.next = node; return node; } } //上一步失败的节点入队 enq(node); return node; }
```

1. 节点加入队尾

```
private Node enq(final Node node) { //自旋锁, 失败重试 for (;;) { //记录队尾元素 Node t = tail; //初始化队列 if (t == null) { //CAS操作, 只有一个线程可以初始化头结点成功, 其余的都要重复执行循环体 if (compareAndSetHead(new Node())) tail = head; } else { //入队的节点指向队尾, 此处会有多个节点指向队尾 node.prev = t; //cas操作, 保证多线程下只有一个节点真正指向队尾节点, 其余失败重试 if (compareAndSetTail(t, node)) { t.next = node; //该循环体唯一退出的操作, 就是入队成功 (否则就要无限重试) return t; } } }
```

1. 将节点加入等待队列后, 先判断是否能够获取锁的资源, 不能则进入挂起状态等待唤醒

```
final boolean acquireQueued(final Node node, int arg) { //锁资源获取失败标记位 boolean failed = true; try { //线程被中断标识 boolean interrupted = false; // 自旋锁等待被唤醒 for (;;) { //获取当前节点的前置节点 final Node p = node.predecessor(); //如果前置节点就是头结点, 则尝试获取锁资源 if (p == head && tryAcquire(arg)) { //将当前节点设置为头结点 setHead(node); p.next = null; //帮助GC //表示锁资源成功获取 failed = false; //返回中断标记, 表示当前节点是被正常唤醒还是被中断唤醒 return interrupted; } //如果没有获取锁成功, 则进入挂起逻辑 //无论被中断或者正常唤醒, 都会进行重新获取锁. 成功则释放, 失败则挂起 // acquireInterruptibly不同在于, 线程阻塞时被中断会抛出异常 if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt()) interrupted = true; } } finally { //获取锁失败处理逻辑 if (failed) cancelAcquire(node); }
```

1. 检测节点状态, 判断是否可进入挂起状态, 同时剔除前边已经取消的节点 `` private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

```
//获取前置节点的waitStatus
int ws = pred.waitStatus;
//如果处于等待唤醒状态, 则返回为true进行堵塞
if (ws == Node.SIGNAL)
    return true;
if (ws > 0) {
    //循环查找没有被取消的前节点, 相当于把之前取消的节点从队列中剔除出去
    do {
        node.prev = pred = pred.prev;
    } while (pred.waitStatus > 0);
    pred.next = node;
} else {
    //将前置节点状态设置Node.SIGNAL, 执行上一步的自旋, 进行堵塞
    compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
}
return false;
```

```
} ``
```

2. 堵塞线程 private final boolean parkAndCheckInterrupt() { //堵塞线程 LockSupport.park(this); //被唤醒之后, 返回中断标记, 即如果是正常唤醒则返回false, 如果是由于中断醒来, 就返回true return Thread.interrupted(); }

3. 获取锁或则加入等待队列失败后的处理

```
`` private void cancelAcquire(Node node) { if (node == null) return;
```

```
node.thread = null;

Node pred = node.prev;
// 跳过所有已经取消的前置节点, 跟上面的那段跳转逻辑类似
while (pred.waitStatus > 0)
    node.prev = pred = pred.prev;

//取得的未取消节点的上一节点
Node predNext = pred.next;
//把当前节点waitStatus置为取消, 这样别的节点在处理时就会跳过该节点
node.waitStatus = Node.CANCELLED;

//如果当前是尾节点, 则直接删除, 即出队
if (node == tail && compareAndSetTail(node, pred)) {
    compareAndSetNext(pred, predNext, null);
} else {
```

```

int ws;
if (pred != head &&
    ((ws = pred.waitStatus) == Node.SIGNAL ||
     (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&
    pred.thread != null) {
    Node next = node.next;
    //如果当前节点的前置节点不是头节点且它后面的节点等待它唤醒（waitStatus小于0），如果当前节点的后继节点没有被取消就把前置节点跟后置节点进行连接
    if (next != null && next.waitStatus <= 0)
        compareAndSetNext(pred, predNext, next);
} else {
    unparkSuccessor(node);
}

node.next = node; // help GC
}
}

```

```

#### 1. 锁的释放

```

public final boolean release(int arg) { // 调用tryRelease方法，尝试去释放锁，由子类具体实现 if (tryRelease(arg)) { Node h = head; //如果队列头节点的状态不是0，那么队列中就可能存在需要唤醒的等待节点。 // if (h != null && h.waitStatus != 0) // 唤醒线程 unparkSuccessor(h); return true; } return false; }

```

#### 1. 线程唤醒

``` private void unparkSuccessor(Node node) {

```

int ws = node.waitStatus;
//如果当前节点的状态小于0，则将该节点置为初始状态，标识节点已完成
if (ws < 0)
    compareAndSetWaitStatus(node, ws, 0);
Node s = node.next;
// 如果下一个节点为null，或者状态是已取消，那么就要寻找下一个非取消状态的节点
if (s == null || s.waitStatus > 0) {
    s = null;
    // 从队列尾向前遍历，直到遍历到node节点
    for (Node t = tail; t != null && t != node; t = t.prev)
        if (t.waitStatus <= 0)
            s = t;
}
if (s != null)
    //唤醒线程
    LockSupport.unpark(s.thread);
}
}

```

```

## 共享锁的实现

1.释放共享锁 ``` private void doAcquireShared(int arg) { // 创建一个共享锁节点 final Node node = addWaiter(Node.SHARED); boolean failed = true; try { boolean interrupted = false; for (;;) { // 获取下一节点 final Node p = node.predecessor(); //前节点为头结点 if (p == head) { //尝试释放共享锁 int r = tryAcquireShared(arg); if (r >= 0) { //获取锁后的唤醒操作 setHeadAndPropagate(node, r); p.next = null; // help GC //如果是被中断唤醒，则设置中断标记为 if ((interrupted) selfInterrupt()); failed = false; return; } } //挂起逻辑同独占锁 if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt()) interrupted = true; } } finally { if (failed) //失败逻辑桶独占锁 cancelAcquire(node); } }

```

1. 线程唤醒 ``` private void setHeadAndPropagate(Node node, int propagate) {

```

//记录头结点以便检查
Node h = head;
//设置当前节点为新的头节点
setHead(node);
//节点满足如下条件则需要进行唤醒线程
if (propagate > 0 || h == null || h.waitStatus < 0 ||
    (h == head) == null || h.waitStatus < 0) {
    Node s = node.next;
    //如果节点s是空或者共享模式节点，那么就要唤醒共享锁上等待的线程
    if (s == null || s.isShared())
        doReleaseShared();
}
}

```

} ```

2. 具体的唤醒逻辑 ``` private void doReleaseShared() {

```

for (;;) {
    Node h = head;
    //从头结点开始唤醒线程
    if (h != null && h != tail) {
        int ws = h.waitStatus;
        //后继节点需要被唤醒
        if (ws == Node.SIGNAL) {
            // 将点h的状态设置成0，如果设置失败，就继续循环，再试一次
            if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                continue; // loop to recheck cases
            //执行唤醒操作
            unparkSuccessor(h);
        }
        //如果后续节点不需要被唤醒，则设置当前节点为PROPAGATE确保以后可以传递
        else if (ws == 0 &&

```



```

        !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
        continue;                // loop on failed CAS
    }
    //头结点没有发生变化, 则表示设置完成, 退出循环
    if (h == head)                // loop if head changed
        break;
}

```

```

}

```

//唤醒node节点的后继节点 private void unparkSuccessor(Node node) { //节点的等待状态 int ws = node.waitStatus; //置零当前线程所在的结点状态, 允许失败 if (ws < 0) compareAndSetWaitStatus(node, ws, 0); //后继节点 Node s = node.next; //后继结点为空或者状态大于0, 则无需处理 if (s == null || s.waitStatus > 0) { s = null; //从尾节点遍历, 找到下一个可唤醒的节点 for (Node t = tail; t != null && t != node; t = t.prev) if (t.waitStatus <= 0) s = t; } if (s != null) //唤醒线程 LockSupport.unpark(s.thread); }

...

JUC之重入锁ReentrantLock

源码解析

lock()

ReentrantLock是在AQS的基础上实现的独占锁。

1. 尝试加锁

```
final void lock() {
    // 检测锁的状态, 如果满足条件, 则将当前锁的拥有者设置为当前线程
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        // 此处逻辑可参考上一篇AQS的文章
        acquire(1);
}
```

1. 非公平锁方式下尝试获取锁

```
final boolean nonfairTryAcquire(int acquires) {
    //获取当前线程
    final Thread current = Thread.currentThread();
    // 获取当前锁的状态
    int c = getState();
    //如果状态为0, 说明当前未有线程占有锁
    if (c == 0) {
        // CAS操作,
        if (compareAndSetState(0, acquires)) {
            // 设置锁的拥有者为当前线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //重入的逻辑
    // 同一线程再次获取锁
    else if (current == getExclusiveOwnerThread()) {
        // 将锁的状态增加1
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

1. 锁的释放

```
public final boolean release(int arg) {
    // 尝试释放锁
    if (tryRelease(arg)) {
        Node h = head;
        // 当头节点不为null以及该节点未被取消的情况下, 唤醒线程
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

1. 具体的释放逻辑

```
protected final boolean tryRelease(int releases) {
    //将当前锁的状态减1
    int c = getState() - releases;
    //如果占有锁的当前线程不是该线程则抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 锁的当前状态为0, 即锁释放完成 (考虑同一根线程加锁两次, 需要释放两次)
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    //设置锁的状态
    setState(c);
    return free;
}
```

1. 公平锁与非公平锁的区别

```
public final boolean hasQueuedPredecessors() {
    Node t = tail;
    Node h = head;
```

```

Node s;
// 通过判断“当前线程”是不是在CLH队列的队首，来返回AQS中是不是有比“当前线程”等待更久的线程
return h != t &&
    ((s = h.next) == null || s.thread != Thread.currentThread());
}

```

lockInterruptibly()

与lock的区别在与如下方法

```

private void doAcquireInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.EXCLUSIVE);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return;
            }
            // 此处，lock只是返回中断状态，而lockInterruptibly则抛出异常
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}

```

tryLock

与lock的区别在于在未获得锁的情况下，并未进行入等待队列的操作，而是直接返回结果。

源码解析（线程等待）

1. await() ``java

```
public final void await() throws InterruptedException {
    // 线程被打断，抛出异常
    if (Thread.interrupted())
        throw new InterruptedException();
    //将节点加入等待序列
    Node node = addConditionWaiter();
    //释放当前线程所占用的lock，在释放的过程中会唤醒同步队列中的下一个节点
    // 此处很好理解，如果不释放所占用的锁，则由于condition会堵塞当前线程，导致，其他
    //线程永远获取不到锁，也就导致了死锁。
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    //自旋，检测节点是否已经进入堵塞队列
    while (!isOnSyncQueue(node)) {
        //线程挂起
        LockSupport.park(this);
        //线程被打断时退出自旋
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    // 被唤醒后，将进入阻塞队列，等待获取锁，即重入锁的lock
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    //后续没有等待的下一节点，则清除该节点
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

2. addConditionWaiter()

```
``java
//加入等待队列
private Node addConditionWaiter() {
    //condition队尾元素
    Node t = lastWaiter;
    //如果队尾元素已经退出，则清除队列
    if (t != null && t.waitStatus != Node.CONDITION) {
        //清除队列
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    //创建condition节点
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    //如果队尾为空，则说明是空队列
    if (t == null)
        //将节点置为头结点
        firstWaiter = node;
    else
        t.nextWaiter = node;
    // 链接成功，将尾节点置为该节点
    lastWaiter = node;
    return node;
}
```

1. unlinkCancelledWaiters()

```
//将已取消的所有节点清除出队列
private void unlinkCancelledWaiters() {
    Node t = firstWaiter;
    Node trail = null;
    // 从头结点开始遍历，
    while (t != null) {
        Node next = t.nextWaiter;
        // 如果节点的状态不是CONDITION，则说明该节点已取消
        if (t.waitStatus != Node.CONDITION) {
            //断开连接
            t.nextWaiter = null;
            if (trail == null)
                //将下一个节点置为队首
                firstWaiter = next;
            else
                trail.nextWaiter = next;
            if (next == null)
                lastWaiter = trail;
        }
        else
            trail = t;
        t = next;
    }
}
```

2. fullyRelease()

```
final int fullyRelease(Node node) {
    boolean failed = true;
    try {
```

```

//获取锁的状态
int savedState = getState();
//释放锁，即重入锁的释放，锁释放失败则抛出异常
if (release(savedState)) {
    failed = false;
    return savedState;
} else {
    throw new IllegalMonitorStateException();
}
} finally {
    //释放失败
    if (failed)
        node.waitStatus = Node.CANCELLED;
}
}

```

3. isOnSyncQueue ()

```

//节点是否已经转移到阻塞队列
final boolean isOnSyncQueue(Node node) {
    // 如果当前节点的状态是CONDITION 或者没有前置节点，说明还没进入阻塞队列
    if (node.waitStatus == Node.CONDITION || node.prev == null)
        return false;
    //如果有下一节点说明已经进入阻塞队列
    if (node.next != null) // If has successor, it must be on queue
        return true;
    //从尾部节点查找该节点
    return findNodeFromTail(node);
}

```

源码解析（线程唤醒）

唤醒之signal()

```

public final void signal() {
    // 调用 signal 方法的线程必须持有当前的独占锁
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    // 找到头结点
    Node first = firstWaiter;
    if (first != null)
        //唤醒头结点
        doSignal(first);
}

```

```

private void doSignal(Node first) {
    do {
        //自旋：从头结点开始寻找第一个能被唤醒的节点
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
    } while ( !transferForSignal(first) &&
              (first = firstWaiter) != null);
}

```

唤醒之signalAll()

```

public final void signalAll() {
    //同样检测
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        doSignalAll(first);
}

```

1. doSignalAll()

```

private void doSignalAll(Node first) {
    lastWaiter = firstWaiter = null;
    do {
        //自旋，从头到尾，依次唤醒
        Node next = first.nextWaiter;
        first.nextWaiter = null;
        transferForSignal(first);
        first = next;
    } while (first != null);
}

```


guava框架

1.1 限流算法

1. 漏桶算法：利用一个缓冲区，当有请求进入系统时，无论请求的速率如何，都先在缓冲区保存，然后以固定的流速流出缓存区进行处理。漏桶算法的特点是无论外部请求压力如何，漏桶算法总以固定的流速处理数据。漏桶的容器和流出速率是该算法的两个重要参数。
2. 令牌桶算法，是一种反向的漏桶算法。在令牌桶算法中，桶中存放的不再是请求，而是令牌，处理程序只有拿到令牌，才能对请求进行处理。如果没有令牌，那么处理程序要么丢弃请求，要么等待可用的令牌。为了限制流速，该算法在每个单位时间产生一定量的令牌存入桶中。（`RateLimiter`采用的就是此算法）。

Disruptor框架

1.1 消费者模式

Disruptor消费者获取缓冲区数据的几种策略

- **BlockWaitStrategy**: 默认策略。使用`BlockWaitStrategy`和使用`BlockQueue`是非常类似的。都是用锁和条件进行数据的监控和线程的唤醒。因为涉及到线程的切换，`BlockWaitStrategy`策略最节省CPU，但是在高并发下它是性能表现最糟糕的一种等待策略。
- **SleepingWaitStrategy**: 这种策略对CPU的消耗和`BlockWaitStrategy`类似。他会在循环中不断等待数据。它会进行自旋等待，如果不成功，则使用`Thread.yield()`方法让出CPU，并最终使用`LockSupport.parkNanos`（1）进行线程休眠，以确保不占用太多的CPU数据。因此，这种策略对于数据处理可能产生比较高的平均延时。他适合对延时要求不是特别高的场合，好处是它对生产者线程的影响最小，典型的应用场景是异步日志。
- **YieldWaitStrategy**: 个策略用于低延时场合。消费者线程会不断循环监控缓冲区的变化，在循环内部，它会使用`Thread.yield`（）方法让出CPU给别的线程执行时间。如果你需要一个高性能的系统，并且对延时有较为严格的要求，则可以考虑这种策略。使用这种策略，相当于消费者线程变成了一个内部执行了`Thread.yield`（）方法的死循环。因此，你最好有多于消费者线程数量的逻辑CPU数量（这里的逻辑CPU指的是“双核四线程”的四线程，否则整个应用程序都会受到影响）。
- **BusySpinWaitStrategy**: 消费者线程会尽最大努力疯狂监控缓冲区的变化。因此，它会吃掉所有的CPU资源。只有对延迟非常苛刻的场合可以考虑使用它。因为使用它等于开启了一个死循环监控，所以你的物理CPU数量必须大于消费者的线程数。

java8新特性

1.1 lambda

常用函数接口：

- Predicate T->boolean 接收T类型的对象并返回boolean值
 - IntPredicate 接收Integer类型对象
 - LongPredicate 接收Long类型对象
 - DoublePredicate 接收Double类型对象
- Consumer T->void 接收T类型的对象并进行处理
 - IntConsumer 接收Integer类型对象
 - LongConsumer 接收Long类型对象
 - DoubleConsumer 接收Double类型对象
- Function T->R 接收T类型对象，进行处理后变成R类型对象
 - IntFunction 接收Integer类型对象，进行处理后变成R类型对象
 - IntToDoubleFunction
 - IntToLongFunction
 - LongFunction
 - LongToDoubleFunction
 - LongToIntFunction
 - DoubleFunction
 - ToIntFunction 接收T类型对象，返回Integer
 - ToDoubleFunction
 - ToLongFunction
- Supplier ()->T Supplier主要是用来创建对象的。可以将耗时操作放在get里，在程序中，传递是Supplier对象，只有真正调用get方法时才执行运算，这就是所谓的惰性求值。
 - BooleanSupplier
 - IntSupplier
 - LongSupplier
 - DoubleSupplier
- UnaryOperator T->T 接收T类型对象，经过处理后返回T类型对象
 - IntUnaryOperator
 - LongUnaryOperator
 - DoubleUnaryOperator
- BinaryOperator (T,T)->T 对两个T类型对象进行处理返回T类型对象
 - IntBinaryOperator
 - LongBinaryOperator
 - DoubleBinaryOperator
- BiPredicate (L,R)->boolean 对两个不同类型对象进行处理，返回boolean类型
- BiConsumer (T,U)->void 对两个对象进行处理
 - ObjIntConsumer
 - ObjLongConsumer
 - objDoubleConsumer
- BiFunction (T,U)->R 对T,U两个类型的对象进行处理并返回R类型数据
 - ToIntBiFunction
 - ToLongBiFunction
 - ToDoubleBiFunction

1.2 Stream

定义：从支持数据处理操作的源生成元素的序列。

详细定义：

1. 元素序列：就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。集合讲的是数据，流讲的是计算。
2. 源：流会使用一个提供数据的源，如集合，数组或输入/输出资源。
3. 数据处理操作：流的数据处理功能类似于数据库的操作，以及函数式编程语言中的常用操作。流操作可以顺序执行，也可并行执行。

流操作特点：

1. 流水线：很多操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的流水线。流水线的操作可以看作对数据源进行数据库式查询。
2. 内部迭代：与使用迭代显式迭代的集合不同，流的迭代操作是在背后进行的。
3. 流只能遍历一次。

1.3.2.1 stream常用方法

(数值类型的流：IntStream, LongStream)

1. map：转换流，将一种类型的流转换为另外一种流。
2. filter：过滤流，过滤流中的元素。

3. **flatMap**: 拆解流，将流中每一个元素拆解成一个流。
4. **sorted**: 对流进行排序。
5. **limit**: 元素不能超过指定长度。短路操作。
6. **distinct**: 去重。返回一个元素各异（根据流生成的`hashCode`和`equal`的方法实现）的流。
7. **skip(n)**: 返回一个扔掉前`n`个元素的流。
8. **flatMap**: 将一个流中的每个值都转换成另一个流，然后将所有的流链接起来成为一个流，见`simpleStreamTest2`。
9. **anyMatch**: 流中是否有一个元素能匹配给定的谓词。
10. **allMatch**: 类似`anyMatch`，查看流中元素是否都能匹配给定的谓词。
11. **noneMatch**: `allMatch`相反。
12. **findAny**: 返回当前元素的任意元素。
13. **findFirst**: 查找第一个元素。
14. **reduce**: 数值计算。

servlet总结

Spring总结

1.1 核心类DefaultListableBeanFactory

DefaultListableBeanFactory是整个bean加载的核心部分，是spring注册及加载bean的默认实现。以下即DefaultListableBeanFactory的组成结构。

- AliasRegistry: 定义对alias的简单增删改等操作。
- SimpleAliasRegister: 使用map作为alias的缓存，并对接口AliasRegistry进行实现。
- SingletonBeanRegistry: 定义对单例的注册及获取。
- BeanFactory: 定义获取bean及bean的各种属性。
- DefaultSingletonBeanRegistry: 对接口SingletonBeanRegistry各函数实现。
- HierarchicalBeanFactory: 继承BeanFactory，在BeanFactory基础上增加对parentFactory的支持。
- BeanDefinitionRegistry: 定义对BeanDefinition的增删改操作。
- FactoryBeanRegistrySupport: 在DefaultSingletonBeanRegistry的基础上增加对FactoryBean的特殊处理功能。
- ConfigurableBeanFactory: 提供配置Facory的各种方法。
- ListableBeanFactory: 根据各种条件获取bean的配置清单。
- AbstractBeanFactory: 综合FactoryBeanRegistrySupport和ConfigurableBeanFactory的功能。
- AutowireCapableBeanFactory: 提供创建bean、自动注入、初始化以及应用bean的后处理器。
- AbstractAutowireCapableBeanFactory: 综合AbstractBeanFactory并对接口AutowireCapableBeanFactory进行实现。
- ConfigurableListableBeanFactory: BeanFactory配置清单，指定忽略类型及接口等。

1.2 XmlBeanDefinitionReader (xml文件的读取)

- ResourceLoader: 定义资源加载器，主要应用于根据给定的资源文件地址返回对应的resource。
- BeanDefinitionReader: 主要定义资源文件读取并转换为BeanDefinition的各个功能。
- EnvironmentCapable: 获取Environment方法
- DocumentLoader: 定义从资源文件加载到转换为Document的功能
- AbstractBeanDefinitionReader: 对EnvironmentCapable、BeanDefinitionReader的功能进行实现。
- BeanDefinitionDocumentReader: 定义读取Document并注册BeanDefinition功能。
- BeanDefinitionParserDelegate: 定义解析Element的各种方法。

1.3 spring对于循环依赖的解决

- 构造器循环依赖: 表示通过构造器注入构成的循环依赖。此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException
- setter循环依赖: 表示通过setter注入方式构成的循环依赖。对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造注入但未完成其他步骤（如setter注入）的bean来完成的，而且只能解决单例作用域的bean循环依赖。通过提前暴露一个单例工厂方法，从而使其他的bean能引用到该bean。通过“setAllowCircularReferences（false）”来禁用循环依赖
- prototype范围的依赖处理: 对于“prototype”作用域bean，spring容器无法完成依赖注入，因为spring容器不进行缓存“prototype”作用域的bean，因此无法提前暴露一个创建中的bean。

1.4 spring生命周期

- 容器启动后，对bean进行初始化。
- 按照的bean的定义，注入属性
- 检测该对象是否实现了Aware接口，并将Aware实例注入到bean中
- 实现BeanPostProcessor接口，进行一些自定义的前置方法处理。
- 调用自定义的初始化方法。如postConstruct，init-method等等
- 实现BeanPostProcessor接口，进行一些自定义的后置方法处理。
- 容器关闭后，如果bean实现了DisposableBean，则会回调该接口的destroy（）方法
- 通过给destroy-method指定函数，可以在bean销毁前执行指定的逻辑

SpringMVC

1.1 springMVC执行流程

- User向服务器发送request,前端控制Servlet DispatcherServlet捕获;
- DispatcherServlet对请求URL进行解析,调用HandlerMapping获得该Handler配置的所有相关的对象,最后以HandlerExecutionChain对象的形式返回.
- DispatcherServlet 根据获得的Handler,选择一个合适的HandlerAdapter.
- 提取Request中的模型数据,填充Handler入参,开始执行Handler (Controller)
- Handler执行完成后,返回一个ModelAndView对象到DispatcherServlet
- 根据返回的ModelAndView, 选择一个适合的ViewResolver
- ViewResolver 结合Model和View, 来渲染视图
- 将渲染结果返回给客户端。

NIO

1.1 TCP的三次握手，四次挥手

三次握手：

- 第一次握手(SYN=1, seq=x)，发送完毕后，客户端进入 SYN_SEND 状态
- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1)，发送完毕后，服务器端进入 SYN_RCVD 状态。
- 第三次握手(ACK=1, ACKnum=y+1)，发送完毕后，客户端进入 ESTABLISHED 状态，当服务器端接收到这个包时，也进入 ESTABLISHED 状态，TCP 握手，即可以开始数据传输。

四次挥手：

- 第一次挥手(FIN=1, seq=a)，发送完毕后，客户端进入 FIN_WAIT_1 状态
- 第二次挥手(ACK=1, ACKnum=a+1)，发送完毕后，服务器端进入 CLOSE_WAIT 状态，客户端接收到这个确认包之后，进入 FIN_WAIT_2 状态
- 第三次挥手(FIN=1, seq=b)，发送完毕后，服务器端进入 LAST_ACK 状态，等待来自客户端的最后一个ACK。
- 第四次挥手(ACK=1, ACKnum=b+1)，客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 TIME_WAIT状态，等待了某个固定时间（两个最大段生命周期，2MSL, 2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 CLOSED 状态。服务器端接收到这个确认包之后，关闭连接，进入 CLOSED 状态。

算法总结

| 排序方法 | 时间复杂度 | 空间复杂度 |
|--------|----------------|----------------|
| 直接插入排序 | $O(n^2)$ | $O(1)$ |
| 希尔排序 | $O(n^2)$ | $O(1)$ |
| 直接选择排序 | $O(n^2)$ | $O(1)$ |
| 堆排序 | $O(n\log_2 n)$ | $O(1)$ |
| 冒泡排序 | $O(n^2)$ | $O(1)$ |
| 快速排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ |
| 归并排序 | $O(n\log_2 n)$ | $O(1)$ |
| 基数排序 | $O(d(r+n))$ | $O(rd+n)$ |

冒泡排序

```
/**冒泡排序：
 * 第一轮：将数组下标为0的数值与 [1,n-1]数值进行比较，遇到比下标为0的数值大的则进行交换，第一轮结束后，最大值就防止到第一位
 * 同理，进行下一轮比较
 */
public class BubbleSort {

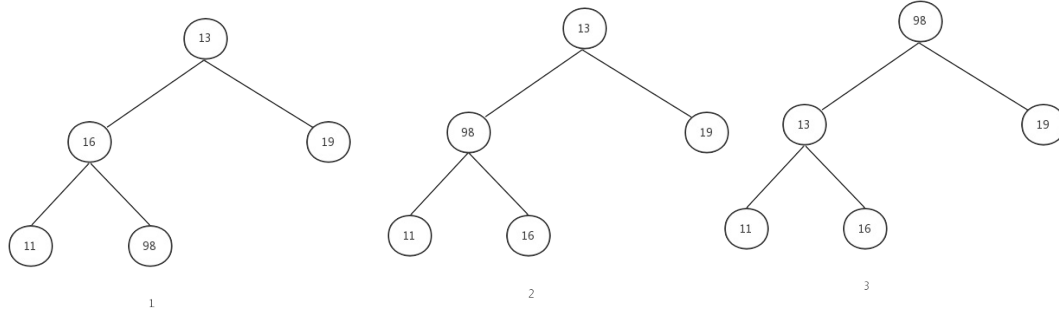
    public static void sort(int arr[], boolean asc) {
        //当比较到arr.length - 1，整个排序结束
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (asc) {
                    //升序
                    if (arr[i] > arr[j]) {
                        exchange(arr, i, j);
                    }
                } else {
                    //降序
                    if (arr[i] < arr[j]) {
                        exchange(arr, i, j);
                    }
                }
            }
        }
    }

    private static void exchange(int arr[], int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] array = new int[]{1, 5, 6, 8, 9, 4, 3, 3, 3, 4, 5, 6};
        System.out.println("排序前: " + Arrays.toString(array));
        sort(array, true);
        System.out.println("升序后: " + Arrays.toString(array));
        sort(array, false);
        System.out.println("降序后: " + Arrays.toString(array));
    }
}
```


堆排序

数组: { 13, 16, 19, 11, 98 } lastIndex=4



- 1.找到第一个非叶子节点 $(lastIndex-1)/2$ ，图中即为16的节点。
- 2.将该节点与左右两个节点依次比较，如果小于子节点，则进行交换。
- 3.对第二个非叶子节点进行处理，即为图上的13节点，重复步骤2。
- 4.此时最大值以放置在根节点中，将根节点与尾节点进行交换（98和16进行交换）。
- 5.继续下次遍历的时候从lastIndex-1，重复以上步骤

```
/**
 * 堆排序
 * 1.构建大根堆（小根堆），找到第一个非叶子节点，从左至右，从下至上进行调整，将最大值放置到父节点处
 * 2.继续对第二个，第三个非叶子节点进行处理，直到到达栈顶，此时的栈顶即为最大值
 * 3.将栈顶元素和最末尾的数进行交换，继续下一次的建堆
 */
public class HeapSort {

    public static void heapSort(int[] a) {
        System.out.println("开始排序");
        int arrayLength = a.length;
        //循环建堆
        for (int i = 0; i < arrayLength - 1; i++) {
            //建堆，每次建堆都会将最大值置为栈顶
            buildMaxHeap(a, arrayLength - 1 - i);
            //交换堆顶和最后一个元素
            swap(a, 0, arrayLength - 1 - i);
            //System.out.println(Arrays.toString(a));
        }
    }

    private static void swap(int[] data, int i, int j) {
        int tmp = data[i];
        data[i] = data[j];
        data[j] = tmp;
    }

    //叶子节点是指没有子节点的节点
    //对data数组从0到lastIndex建大顶堆
    private static void buildMaxHeap(int[] data, int lastIndex) {
        //从lastIndex处节点（最后一个节点）的父节点开始（此节点的父节点为第一个非叶子节点）
        //从该节点往上，每个节点都为非叶子节点
        for (int i = (lastIndex - 1) / 2; i >= 0; i--) {
            //k保存正在判断的节点
            int k = i;
            //如果当前k节点的子节点存在，左节点等于k*2+1,右节点等于k*2+2
            while (k * 2 + 1 <= lastIndex) {
                //k节点的左子节点的索引
                int biggerIndex = 2 * k + 1;
                //如果biggerIndex小于lastIndex，即biggerIndex+1代表的k节点的右子节点存在
                if (biggerIndex < lastIndex) {
                    //若果右子节点的值较大
                    if (data[biggerIndex] < data[biggerIndex + 1]) {
                        //biggerIndex总是记录较大子节点的索引
                        biggerIndex++;
                    }
                }
                //如果k节点的值小于其较大的子节点的值
                if (data[k] < data[biggerIndex]) {
                    //交换他们
                }
            }
        }
    }
}
```

```

        swap(data, k, biggerIndex);
        //将biggerIndex赋予k.
        //开始while循环的下一循环, 重新保证k节点的值大于其左右子节点的值
        k = biggerIndex;
    } else {
        break;
    }
}
}
}

public static void printArray(int arr[]) {
    for (int k = 0; k < arr.length; k++) {
        System.out.print(arr[k] + "\t");
    }
}

public static void main(String[] args) {
    int[] data = {543, 23, 45, 65, 76, 1, 456, 7, 77, 88, 3, 9};
    System.out.print("数组排序前: ");
    printArray(data);
    System.out.print("\n");
    heapSort(data);
    System.out.print("排序后: ");
    printArray(data);
}
}

```

插入排序

```
/**
 * 插入排序：
 * 插入排序(Insert Sort)将待排序的数组分为2部分：有序区，无序区。其核心思想是每次取出无序区中的第一个元素，
 * 插入到有序区中。 有序与无序区划分，就是通过一个变量标记当前数组中，前多少个元素已经是局部有序了。
 */
public class InsertSort {

    public static void sort(int arr[], boolean asc) {
        //设定有序区的起始位置
        int orderlyIndex = 0;
        for (int i = orderlyIndex + 1; i < arr.length; i++) {
            int temp = arr[i]; //记录无序区中的第一个元素值
            int insertIndex = i; //初始设置位置为自己的位置
            for (int j = orderlyIndex; j >= 0; j--) {
                if (asc) {
                    //升序，所有比temp大的值都右移
                    if (temp < arr[j]) {
                        //右移
                        arr[j + 1] = arr[j];
                        //每移动一次，插入的索引减1
                        insertIndex--;
                    } else {
                        //有序区当前位置元素<=无序区第一个元素，那么之前的元素都会<=，不需要继续比较
                        break;
                    }
                } else {
                    if (temp > arr[j]) {
                        arr[j + 1] = arr[j];

                        insertIndex--;
                    } else {
                        break;
                    }
                }
            }
            orderlyIndex++;
            arr[insertIndex] = temp;
        }
    }

    public static void main(String[] args) {
        int[] array = new int[]{1, 5, 6, 8, 9, 4, 3, 3, 3, 4, 5, 6};
        System.out.println("排序前: " + Arrays.toString(array));
        sort(array, true);
        System.out.println("升序后: " + Arrays.toString(array));
        sort(array, false);
        System.out.println("降序后: " + Arrays.toString(array));
    }
}
```

归并排序

```
/**
 * 归并排序：
 * 将一个数组拆分成两半，分别对每一半进行排序，然后使用合并(merge)操作，把两个有序的子数组组合成一个整体的有序数组。
 * 我们可以把一个数组刚开始先分成两，也就是2个1/2，之后再每一半分成两半，也就是4个1/4，
 * 以此类推，反复的分隔数组，直到得到的子数组中只包含一个数据项，这就是基值条件，只有一个数据项的子数组肯定是有顺序的。
 */
public class MergeSort {

    public static void mergeSort(int arr[], int left, int right) {
        if (left < right) {
            int mid = (right + left) / 2;
            //递归分解左边数组
            mergeSort(arr, left, mid);
            //递归分解右边数组
            mergeSort(arr, mid + 1, right);
            //分解到左右各只有一个元素时，进行合并
            merge(arr, left, right);
        }
    }

    private static void merge(int arr[], int startIndex, int endIndex) {
        //记录数组中间位置
        int mid = (endIndex + startIndex) / 2;
        //创建一个临时数组
        int[] temp = new int[endIndex - startIndex + 1];
        //左侧数组起始位置
        int leftStartIndex = startIndex;
        //右侧数组起始位置
        int rightStartIndex = mid + 1;
        //记录临时数组索引位置
        int count = 0;
        //对两个数组进行归并
        while (leftStartIndex <= mid && rightStartIndex <= endIndex) {
            if (arr[leftStartIndex] <= arr[rightStartIndex]) {
                temp[count++] = arr[leftStartIndex++];
            } else {
                temp[count++] = arr[rightStartIndex++];
            }
        }
        while (leftStartIndex <= mid) {
            temp[count++] = arr[leftStartIndex++];
        }
        while (rightStartIndex <= endIndex) {
            temp[count++] = arr[rightStartIndex++];
        }
        //复制临时数组到原数组中
        System.arraycopy(temp, 0, arr, startIndex, count);
    }

    public static void printArray(int arr[]) {
        for (int k = 0; k < arr.length; k++) {
            System.out.print(arr[k] + "\t");
        }
    }

    public static void main(String[] args) {
        int[] data = {543, 23, 45, 65, 76, 1, 456, 7, 77, 88, 3, 9};
        System.out.print("数组排序前: ");
        printArray(data);
        System.out.print("\n");
        mergeSort(data, 0, data.length - 1);
        System.out.print("归并排序后: ");
        printArray(data);
    }
}
```

快速排序

```
/**
 * 快速排序 (6 1 2 7 9 3 4 5 10 8)
 * 1.选择基准值,比如选第一个6为基准值
 * 2.从i=1 开始向右寻找第一个比6大的数, j从最后一个值往前找,找到第一个比6小的数,交换i,j
 * 3.i,j继续往前走,继续做类似第二步的交换
 * 4.知道i和j相遇了,或者i>j了,则停止,这时候基准值左边都比基准值小,右边都比基准值大
 * 5.此时基准值位于j所处位置,将已j分成两个数组,继续进行排序
 */
public class QuickSort {
    public static void sort(int array[], int leftIndex, int rightIndex) {
        if (leftIndex >= rightIndex) {
            return;
        }
        int partitionIndex = doSort(array, leftIndex, rightIndex);

        sort(array, leftIndex, partitionIndex - 1);
        sort(array, partitionIndex + 1, rightIndex);
    }

    public static int doSort(int[] arr, int start, int end) {
        int pivot = getPivot(arr, start, end);
        int left_pointer = start - 1;
        int right_pointer = end + 1;
        while (true) {
            //left_pointer当遇到比基准值大的元素,停下来
            while (arr[++left_pointer] < pivot) {
            }
            //right_pointer当遇到比基准值小的元素,停下来
            while (arr[--right_pointer] > pivot) {
            }
            if (left_pointer >= right_pointer) {
                break;
            }
            int temp = arr[left_pointer];
            arr[left_pointer] = arr[right_pointer];
            arr[right_pointer] = temp;
        }
        return right_pointer;
    }
}

/**
 * 获取基准值
 */
private static int getPivot(int array[], int start, int end) {
    return array[start];
}

public static void main(String[] args) {
    // int[] arr = {3,4,2,0,4,7,9,6,5,8};
    int[] arr = {3, 3, 3, 3, 3, 4, 7, 2, 6, 5, 8};
    System.out.println("排序前数组:" + Arrays.toString(arr));
    sort(arr, 0, arr.length - 1);
    System.out.println("排序前数后:" + Arrays.toString(arr));
    Arrays.sort(arr);
}
}
```

选择排序

```
/**
 * 选择排序，它的工作原理是每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。
 * 与冒泡排序不同的是，选择排序不会一遇到两个数字的顺序不对时，立马交换其位置，而是记录其下标，等该轮结束后进行交换
 */
public class SelectionSort {
    public static void sort(int arr[], boolean asc) {
        //当比较到arr.length - 1，整个排序结束
        for (int i = 0; i < arr.length - 1; i++) {
            int index = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (asc) {
                    //升序
                    if (arr[index] > arr[j]) {
                        index = j;
                    }
                } else {
                    //降序
                    if (arr[index] < arr[j]) {
                        index = j;
                    }
                }
            }
            exchange(arr, i, index);
        }
    }

    private static void exchange(int arr[], int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] array = new int[]{1, 5, 6, 8, 9, 4, 3, 3, 3, 4, 5, 6};
        System.out.println("排序前: " + Arrays.toString(array));
        sort(array, true);
        System.out.println("升序后: " + Arrays.toString(array));
        sort(array, false);
        System.out.println("降序后: " + Arrays.toString(array));
    }
}
```

希尔排序

```
/**
 * 希尔排序：
 * 将数的个数设为n，取奇数k=n/2，将下标差值为k的书分为一组，构成有序序列。
 * <p>
 * 再取k=k/2，将下标差值为k的书分为一组，构成有序序列。
 * <p>
 * 重复第二步，直到k=1执行简单插入排序。
 */
public class SheelSort {

    public static void sheelSort(int[] a) {
        int d = a.length;
        while (d != 0) {
            d = d / 2;

            for (int x = 0; x < d; x++) { //分的组数
                for (int i = x + d; i < a.length; i += d) { //组中的元素，从第二个数开始
                    int j = i - d; //j为有序序列最后一位的位数
                    int temp = a[i]; //要插入的元素
                    for (; j >= 0 && temp < a[j]; j -= d) { //从后往前遍历，满足交换条件才会执行
                        a[j + d] = a[j]; //向后移动d位
                    }
                    //根据交换次数来决定temp放到什么位置
                    a[j + d] = temp;
                }
            }
        }
    }

    public static void printArray(int arr[]) {
        for (int k = 0; k < arr.length; k++) {
            System.out.print(arr[k] + "\t");
        }
    }

    public static void main(String[] args) {
        //int[] data = {543, 23, 45, 65, 76, 1, 456, 7, 77, 88, 3, 9};
        int[] data = {7, 6, 4, 8, 9};
        System.out.print("数组排序前: ");
        printArray(data);
        System.out.print("\n");
        sheelSort(data);
        System.out.print("排序后: ");
        printArray(data);
    }
}
```

设计模式

1.1 策略模式

在策略模式（**Strategy Pattern**）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 **context** 对象。策略对象改变 **context** 对象的执行算法。

优点：

- 算法可以自由切换。
- 避免使用多重条件判断。
- 扩展性良好。

缺点：

- 策略类会增多。
- 所有策略类都需要对外暴露。

1.2 单例模式

单例模式总共有5种实现方式，分别如下

- 饿汉式(线程安全，调用效率高，但是不能延时加载)
- 懒汉式(线程安全，调用效率不高，但是能延时加载)
- **Double CheckLock**实现单例：**DCL**也就是双重锁判断机制（由于JVM底层模型原因，偶尔会出问题，不建议使用）：
- 静态内部类实现模式（线程安全，调用效率高，可以延时加载）
- 枚举类（线程安全，调用效率高，不能延时加载，可以天然的防止反射和反序列化调用）

单例模式

1. 饿汉式

```
public class Singleton{
    private static Singleton instance = new Singleton();
    private Singleton(){}
    public static Singleton getInstance(){
        return instance;
    }
}
```

1. 懒汉式

```
public class Singleton {

    //类初始化时，不初始化这个对象(延时加载，真正用的时候再创建)
    private static Singleton instance;

    //构造器私有化
    private Singleton(){}

    //方法同步，调用效率低
    public static synchronized Singleton getInstance(){
        if(instance==null){
            instance=new Singleton();
        }
        return instance;
    }
}
```

1. 双重检查锁

```
public class Singleton {
    private volatile static Singleton Singleton;

    private Singleton() {
    }

    public static Singleton newInstance() {
        if (Singleton == null) {
            synchronized (Singleton.class) {
                if (Singleton == null) {
                    Singleton = new Singleton();
                }
            }
        }
        return Singleton;
    }
}
```

1. 静态内部类

```
public class Singleton {

    private static class SingletonClassInstance{
        private static final Singleton instance=new Singleton();
    }

    private Singleton(){}

    public static Singleton getInstance(){
        return SingletonClassInstance.instance;
    }
}
```

1. 枚举类

```
public enum Singleton {

    //枚举元素本身就是单例
    INSTANCE;

    //添加自己需要的操作
    public void singletonOperation(){
    }
}
```


分布式

分布式系统：是一个硬件或软件分布在不同的网络计算机上，彼此之间只通过消息传递进行通信和协调的系统。

特点：

1. 分布性：分布式系统中的多台计算机都在空间上随意分布。
2. 对等性：分布式系统中的计算机没有主从之分，组成分布式系统的所有计算机节点都是对等的。
3. 并发性：分布式系统的多个节点，可能并发的操作一些共享的数据。
4. 缺乏全局时钟：在分布式系统中，很难定义两个事件究竟谁先谁后，原因就是分布式系统缺乏一个全局的时钟序列控制。
5. 故障总是会发生：组成分布式系统的所有计算机，都有可能发生任何形式的故障。

分布式事务

1.1 ACID传统数据库

- **Atomicity**原子性：一个事务中所有操作都必须全部完成，要么全部不成功。
- **Consistency**一致性：在事务开始或结束时，数据库应该在一致状态。
- **Isolation**隔离层：并发的事务是相互隔离的，一个事务的执行不能被其他事务干扰。
 - **Read Uncommitted**：允许脏读取。可以读取其他事务未提交的数据。
 - **Read Committed**：只允许获取已经被提交的数据。
 - **Repeatable Read**：可重复读取。就是保证在事务处理过程中，多次读取同一数据时，其值和事务开始时刻是一致的。可能会出现幻影数据，即在不同时间段读取同一数据项，可能出现不一致的结果。
 - **Serializable**：要求所有事务都被串行执行。
- **Durability**. 一旦事务完成，就不能返回。

1.2 CAP理论

- **Partition tolerance**:分区容错。分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障。
- **Consistency**：一致性。是指数据在多个副本之间能够保持一致的特性。
- **Availability**：可用性。系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

在分布式系统中，只可能满足CP或者AP。如果要保证一致性，那么写的操作必然需要加锁，这样就不满足可用性。如果要保证可用性，那么在通信失败的情况下，无法满足一致性。所以在分布式系统下，需要对一致性和可用性进行取舍。

1.3 BASE理论

- **基本可用（Basically Available）**：是指在分布式系统出现故障的时候，允许损失部分可用性，即保证核心可用。（服务降级）。
 - 响应时间上的损失
 - 功能上的损失
- **软状态（Soft State）**：允许系统存在中间状态，该中间状态不会影响系统整体可用性。即允许系统在不同节点的数据副本之间进行同步的过程存在延时。
- **最终一致性**：是指系统中的所有数据副本经过一段时间后，最终能达到一致的状态。

实际工程实践的变种

- **因果一致性**：**A**更新完数据项通知**B**，那么**B**对于该数据项进行更新操作时，必须基于**A**更新后的最新值。
- **读己之所写**：进程**A**更新一个数据项后，它总能访问到更新过的最新值。
- **会话一致性**：将系统数据的访问过程框在一个会话中，保证在同一个有效的会话中实现“读己之所写”的一致性。
- **单调读一致性**：一个进程从系统中读取出一个数据项的某个值后，那么系统对于该进程后续的任何数据访问都不应该返回旧的值。
- **单调写一致性**：一个系统需要能够保证来自同一进程的写操作被顺序的执行

1.4 两阶段提交（2pc）

第一阶段：准备阶段。

- 协调者节点向所有参与者节点询问是否可以执行提交操作(**vote**)，并开始等待各参与者节点的响应。
- 参与者节点执行询问发起为止的所有事务操作，并将**Undo**信息和**Redo**信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）
- 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个“同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个“中止”消息。

第二阶段：提交阶段。

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(**Rollback**)消息；否则，发送提交(**Commit**)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源)

- 当协调者节点从所有参与者节点获得的相应消息都为“同意”时: 1) 协调者节点向所有参与者节点发出“正式提交(**commit**)”的请求。 2) 参与者节点正式完成操作，并释放整个事务期间内占用的资源。 3) 参与者节点向协调者节点发送“完成”消息。 4) 协调者节点受到所有参与者节点反馈的“完成”消息后，完成事务。
- 当任一参与者节点在第一阶段返回的响应消息为“中止”或协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时: 1) 协调者节点向所有参与者节点发出“回滚操作(**rollback**)”的请求。 2) 参与者节点利用之前写入的**Undo**信息执行回滚，并释放在整个事务期间内占用的资源。 3) 参与者节点向协调者节点发送“回滚完成”消息。 4) 协调者节点受到所有参与者节点反馈的“回滚完成”消息后，取消事务。

缺点：

1. 同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
2. 单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。
3. 数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送**commit**请求之后，发生了局部网络异常或者在发送**commit**请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了**commit**请求。而在这部分参与者接到**commit**请求之后就会执行**commit**操作。但是其他部分未接到**commit**请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。
4. 协调者再发出**commit**消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否已经被提交。

1.5 三阶段提交（3pc）

在二阶段的基础上：

- 引入超时机制。同时在协调者和参与者中都引入超时机制。

- 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

- **CanCommit阶段**

协调者向参与者发送**commit**请求，参与者如果可以提交就返回成功响应，否则返回失败响应。

- 事务询问 协调者向参与者发送**CanCommit**请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。
- 响应反馈 参与者接到**CanCommit**请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回**Yes**响应，并进入预备状态。否则反馈**No**。

- **PreCommit阶段**

协调者根据参与者的反应情况来决定是否可以记性事务的**PreCommit**操作。

- 发送预提交请求 协调者向参与者发送**PreCommit**请求，并进入**Prepared**阶段。
- 事务预提交 参与者接收到**PreCommit**请求后，会执行事务操作，并将**undo**和**redo**信息记录到事务日志中。
- 响应反馈 如果参与者成功的执行了事务操作，则返回**ACK**响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了**No**响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

- 发送中断请求 协调者向所有参与者发送**abort**请求。
- 中断事务 参与者收到来自协调者的**abort**请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

1. **doCommit阶段**

执行提交

- i. 发送提交请求 协调接收到参与者发送的**ACK**响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送**doCommit**请求。
- ii. 事务提交 参与者接收到**doCommit**请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
- iii. 响应反馈 事务提交完之后，向协调者发送**Ack**响应。
- iv. 完成事务 协调者接收到所有参与者的**ack**响应之后，完成事务。

中断事务（协调者没有接收到参与者发送的**ACK**响应）

1. 发送中断请求 协调者向所有参与者发送**abort**请求
2. 事务回滚 参与者接收到**abort**请求之后，利用其在阶段二记录的**undo**信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
3. 反馈结果 参与者完成事务回滚之后，向协调者发送**ACK**消息
4. 中断事务 协调者接收到参与者反馈的**ACK**消息之后，执行事务的中断

注：在**doCommit**阶段，如果参与者无法及时接收到来自协调者的**doCommit**或者**reboot**请求时，会在等待超时之后，会继续进行事务的提交。

- 发送提交请求 协调接收到参与者发送的**ACK**响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送**doCommit**请求。
- 事务提交 参与者接收到**doCommit**请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
- 响应反馈 事务提交完之后，向协调者发送**Ack**响应。
- 完成事务 协调者接收到所有参与者的**ack**响应之后，完成事务。

中断事务

- 1.发送中断请求 协调者向所有参与者发送**abort**请求
- 2.事务回滚 参与者接收到**abort**请求之后，利用其在阶段二记录的**undo**信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
- 3.反馈结果 参与者完成事务回滚之后，向协调者发送**ACK**消息
- 4.中断事务 协调者接收到参与者反馈的**ACK**消息之后，执行事务的中断。 在**doCommit**阶段，如果参与者无法及时接收到来自协调者的**doCommit**或者**reboot**请求时，会在等待超时之后，会继续进行事务的提交。

1.6 Paxos算法

Zookeeper

1.1 Zookeeper简介

Zookeeper是一个开源的分布式协调服务，是一个典型的分布式数据一致性的解决方案，分布式应用程序可以基于此实现诸如数据发布/订阅，负载均衡，命名服务，分布式协调/通知，集群管理，**Matser**选举，分布式锁和分布式队列等功能。

1.2 Zk特性

- 顺序一致性：从同一个客户端发起的事务请求，最终将会严格地按照其发起顺序被应用到**Zk**中去。
- 原子性：所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群的所有机器都成功应用了某个事务，要么都没有应用。
- 单一视图：无论客户端连接是哪个**zk**服务器，其看到的服务端数据模型都是一致的。
- 可靠性：一旦服务端成功应用了一个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会被一直保留下来，除非有另外一个事务对其进行了变更。
- 实时性：在一定的时间内，客户端最终一定能够从服务端上读取到最新的数据状态。

1.3 zk集群

zk集群中有三种角色，**Leader**，**Follower**，**Observer**。**zk**集群中的所有机器通过一个**Leader**选举过程来选定一台名为"**Leader**"的机器，**Leader**服务器为客户端提供读和写服务。**Follower**，**Observer**都能提供读服务，区别在于**Observer**不参加写操作的"过半写成功"策略。因此**Observer**可以在不影响写性能的情况下提升集群的读性能。

1.4 会话

客户端和服务端之间建立一个**TCP**长连接，通过这个连接，客户端可以通过心跳检测与服务端保持有效的会话，也能够向服务端发送请求并接受响应，同时接收来自服务器的**Watch**事件通知。**Session**的**sessionTimeout**值用来设置一个客户端会话的超时时间。在服务器发生故障时，只要在**sessionTimeout**规定的时间内能够重新连接到集群中的任意一台服务器，那么之前创建的会话仍然有效。

1.5 数据节点（Znode）

zk将所有数据都存在内存中，数据模型是一颗树，有斜杠（/）进行分割路径。

在**zk**中，**ZNode**分为持久节点和临时节点。所谓持久节点是指一旦这个**ZNode**被创建了，除非主动进行**ZNode**的移除操作，否则这个**ZNode**将一直保存在**zk**上。

临时节点，生命周期与客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。**zk**还允许用户为每个节点添加一个特殊的属性：**SEQUENTIAL**。一旦节点被标记这个属性，那么这个节点在创建的时候，**zk**会自动在其节点后面追加一个整形数字，这个整形数字是一个由父节点维护的自增数字。

1.6 Watch

zk允许用户在指定节点上注册一些**watch**，并且在一些特定事件触发的时候，**zk**会将事件通知到感兴趣的客户端上去，该机制是**zk**实现分布式协调服务的重要特性。

1.7 权限

- **CREATE**：创建子节点的权限。
- **READ**：获取节点数据好子节点列表的权限。
- **WRITE**：更新节点数据的权限。
- **DELETE**：删除子节点的权限。
- **ADMIN**：设置节点的**ACL**权限
-

1.8 ZAB协议

定义：所有事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器称为**Leader**服务器，而余下的其他服务器都是**Follower**服务器。**LEADER**服务器负责将一个客户端请求转换成一个事务**Proposal**（提案），并将该**Proposal**分发个集群中所有**Follower**服务器。之后**Leader**服务器需要等待所有**Follower**的反馈，一旦超过半数的**Follower**服务器进行正确的反馈后，那么**Leader**就会再次向所有的**Follower**服务器分发**Commit**消息，要求将**Proposal**进行提交。

- 崩溃恢复：当**Leader**服务器出现故障的时候，**ZAB**协议就会进入恢复模式并选举新的**Leader**服务器。当选举产生新的**Leader**服务器后，同时集群中已经有过半机器与该**Leader**服务器完成了状态同步后，**ZAB**协议就会退出恢复模式。当遵守**ZAB**协议的服务器新加入时，会自动进入恢复模式，与**Leader**服务器进行数据同步。

ZAB协议需要确保那些已经在**Leader**服务器上提交的事务最终被所有服务器都提交。

ZAB协议需要确保丢弃那些只在**Leader**服务器上被提出的事务。

为了满足以上两种情况，**ZAB**协议设计了如下的**Leader**选举算法：保证新选举出来的**Leader**服务器拥有集群中所有机器最高编号（即**ZXID**最大）的事务**Proposal**，那么就可以保证新选举出来的**Leader**一定具有所有已经提交的提案。

- 消息广播：**Leader**服务器在接收到客户端的事务请求后，会生成对应事务提案并发起一轮广播协议；而如果集群中其他机器接收到客户端的事务请求，那么这些非**Leader**服务器会首先将这个事务请求转发给**Leader**服务器。在整个消息广播过程中，**Leader**服务器会为每个事务请求生成对应的**Proposal**来进行广播，并且在广播**Proposal**之前，**Leader**会为此事务**Proposal**分配一个全局单调递增的唯一ID，我们称为事务ID（即**ZXID**）。由于**ZAB**协议需要保证每一个消息严格的因果关系，因此必须将每一个事务**Proposal**按照其**ZXID**的先后顺序进行排序与处理。

在广播过程中，**Leader**服务器会为每一个**Follower**服务器分配一个单独的队列，然后将需要广播的事务**Proposal**依次放入到这个队列中去，并且根据**FIFO**策略进行消息发送。每一个**Follower**服务器接收到**Proposal**后，都会首先将其以事务日志的形式写入到本地磁盘中去，并且在成功写入后反馈给**Leader**服务器一个**Ack**响应。当**Leader**服务器收到超过半数**Follower**的**Ack**响应后，就会广播一个**Commit**消息给所有的**Follower**服务器以通知其进行事务提交，同时**Leader**也会完成对事务的提交。

ZXID是一个64位的数字，其中低32位可以看做一个简单的单调递增的计数器，**Leader**在产生一个新的事务**Proposal**的时候，都会对计数器加1。而高32位代表**Leader**周期的**epoch**的编号。每当选举产生一个新的**Leader**服务器，都会从这个**Leader**服务器上取出其本地日志最大事务的**Proposal**的**ZXID**，并从该**ZXID**解析出对应的**epoch**值，然后对其进行加1操作，并以此作为新的**epoch**，并将低32位位置0来开始生成新的**ZXID**。

1.8 ZAB与Paxos算法的区别联系

相同点：

1. 两个都存在一个类似**Leader**进程的角色，由其负责协调多个**follower**进程的运行。

2. **Leader**进程都会等待超过半数的**Follower**做出正确的反馈后，才会将一个提案进行提交
3. 在**ZAB**协议中，每个**Proposal**中都包含一个**epoch**值，用来代表当前的**Leader**周期，在**Proxs**算法中，同样存在这样一个标识，**Ballot**。

不同点：

在**Paxos**算法中，一个新选举的主进程会进行两个阶段的工作。第一阶段被称为读阶段，在这个阶段中，这个新的主进程会通过和其它进程进行通信的方式来收集上一个主进程提出的方案，并将它们提交。第二阶段被称为写阶段，在这个阶段中，当前主进程开始提出它自己的提案。

ZAB协议在**paxos**的基础上，增加了一个同步的阶段。再同步阶段之前，**ZAB**协议也存在一个和**Paxos**算法中的读阶段类似的过程，称为发现阶段。在同步阶段，新的**Leader**会确保存在过半的**Follower**已经提交了之前**Leader**周期中的所有事务**Proposal**。这一同步阶段的引入，能够有效地保证**Leader**在新的周期提出事务**Proposal**之前，所有的进程都已经完成了对之前所有事务**Proposal**的提交。一旦完成同步阶段，那么**ZAB**就会执行和**Paxos**算法类似的些阶段。

ZAB主要用于构建一个高可用的分布式数据主备系统。

Paxos主要用于构建一个分布式的一致性状态机系统。

1.9 zk的典型应用场景

1. 发布/订阅
2. 负载均衡
3. 命名服务
4. 分布式协调/通知
5. 集群管理
6. Master选举
7. 分布式锁
8. 分布式队列

2.0 Znode类型

持久节点：数据节点从创建后一直存在于**zk**服务器上，直到有删除操作。

持久顺序节点：每一个父节点都会为它的第一级节点维护一份顺序，用于记录每个子节点的先后顺序。

临时节点：生命周期和客户端的会话绑定在一起。客户端会话失效，节点自动清理。

临时顺序节点：

2.1 状态STAT

数据节点的状态信息

| 状态属性 | 说明 |
|----------------|---------------------------------|
| czxid | 数据节点创建时的事务id |
| mzxid | 节点最后一次被更新时的事务id |
| ctime | 创建时间 |
| mtime | 更新时间 |
| version | 数据节点的版本号 |
| cversion | 子节点版本号 |
| aversion | 节点的ACL版本号 |
| ephemeralowner | 创建该临时节点的会话的sessionid。如果是持久节点，则0 |
| dataLength | 数据内容的长度 |
| numChildren | 当前节点的子节点个数 |
| pzxid | 该节点的子节点列表最后一次被修改时的事务id |

2.2 watcher通知状态

| KeeperState | EventType | 触发条件 | 说明 |
|------------------|---------------------|-------------------------------------|---------------------------|
| SyncConnected（3） | None（-1） | 客户端与服务器成功建立会话 | 与服务器处于连接状态 |
| SyncConnected（3） | NodeCreate | Watcher监听的对应数据节点被创建 | |
| SyncConnected（3） | NodeDeleted | watcher监听的对应数据节点被删除 | |
| SyncConnected（3） | NodeDataChanged | Watcher监听的对应数据节点的数据内容发生变更 | |
| SyncConnected（3） | NodeChildrenChanged | Watcher监听的对应数据节点的数据内容发生变更 | |
| Disconnected（0） | None（-1） | 客户端和服务端断开连接 | |
| Expired（-112） | None（-1） | 会话超时 | SessionExpiredException异常 |
| AuthFailed（4） | None（-1） | 通常两种情况：使用错误的scheme进行权限检查，SASL权限检查失败 | AuthFailedException |

watcher的特性

- 一次性：无论客户端还是服务端，一旦一个Watcher被触发，Zookeeper都会将其从对应的存储中移除。所以在使用watcher的时候需要反复注册。这样的设计有效减轻了服务器的压力。

- 客户端串行执行：客户端watcher回调的过程是一个串行同步的过程。
- 轻量：watchedEvent是zk整个watcher通知机制的最小通知单元，这个数据结构只包含三部分：通知状态，事件类型和节点路径。也就是说，watcher的通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

2.3 ACL--保障数据的安全

ACL机制：权限模式（scheme），授权对象（ID）和权限（Permission），通常使用“schema: id:permission”来标识一个有效的ACL信息。

2.3.1 权限模式：scheme

- IP：IP模式通过IP地址粒度来进行权限控制。
- Digest：“username: password”形式的权限标识来进行权限配置，便于区分不同应用来进行权限控制
- world：最开放的权限控制模式。数据节点的访问权限对所有用户开放，所有用户都可以在不进行任何权限校验的情况下操作zk上的数据。特殊的Digest模式，权限标识：“world: anyone”
- Super：超级用户，也是一种特殊的Digest模式。在super模式下，超级用户可以对任意zk上的数据节点进行任何操作。

2.3.2 授权对象：ID

授权对象指的是权限赋予的用户或一个制定实体，例如IP地址或是机器。

2.3.3 权限：Permission

权限是指那些通过权限检查后可以被允许执行的操作。

- CREATE（C）：数据节点的创建权限，允许授权对象在该数据节点下创建子节点。
- DELETE（D）：子节点的删除权限。
- READ（R）：数据节点的读取权限
- WRITE（W）：数据节点的写入权限
- ADMIN（A）：数据节点的管理权限，允许授权对象对该数据节点进行ACL相关的设置操作。

redis

1.1 redis的持久化方式

- **RDB**: 在指定的时间间隔对数据进行快照存储。**Redis**可以通过创建快照来获取存储在内存中的数据在某个时间点上的副本。**redis**创建快照之后, 可以对快照进行备份, 可以将快照复制到其他服务器从而创建相同数据的服务器副本。
- **AOF**: 记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据。

1.2 线程模型

redis内部使用文件事件处理器, 这个文件处理器是单线程的, 所以**redis**才叫做单线程的模型。他采用**IO**多路复用机制同时监听多个**socket**, 根据**socket**上的事件: 来选择对应的事件处理器进行处理。

文件处理器包含四个结构: 多个**socket**, **IO**多路复用程序, 文件事件分派器, 事件处理器(连接应答处理器, 命令请求处理器, 命令回复处理器)。

1.3 redis过期两种删除方式

- 定时删除: **redis**默认每隔100ms就随机抽取一些设置了过期时间的**key**, 检查其是否过期, 如果过期就删除
- 惰性删除: 定期删除可能会导致很多过期的**key**到了时间并没有删除掉。加入你的过期**key**, 考定期删除没有被删除掉, 还停留在内存中, 除非你的系统去查一下那个**key**, 才会被**redis**给删除。

1.4 redis的内存淘汰机制

1. **volatile-lru**: 从以设置过期时间的数据集中挑选最近最少使用的数据淘汰。
2. **volatile-ttl**: 从已设置过期时间的数据集中挑选将要过期的数据淘汰
3. **volatile-random**: 从已设置过期时间的数据集中任意选择数据淘汰
4. **allkeys-lru**: 当内存不足以容纳新写入数据时, 在键空间中, 移除最近最少使用的**key**
5. **allkeys-random**: 从数据集中任意选择数据淘汰
6. **no-eviction**: 禁止驱逐数据, 也就是说当内存不足以容纳新写入数据时, 新写入操作会报错
7. **volatile-lfu**: 从已设置过期时间的数据集中挑选最经常使用的数据淘汰
8. **allkeys-lfu**: 当内存不足以容纳新写入数据时, 在键空间中, 移除最不经常使用的**key**

1.5 redis产生的相关问题

- 缓存穿透: 是指查询一个数据库一定不存在的数据。正常的使用缓存的流程大致时, 数据查询先进行缓存查询, 如果**key**不存在或**key**已经失效, 在对数据库进行查询, 并把查询的对象, 放入缓存。如果数据库查询对象为空, 则不放进缓存。解决方案, 对空值也做缓存, 缓存周期设置较短一些。或此采用布隆过滤器, 将所有可能存在的数据哈希到一个足够大的**bitmap**中。
- 缓存雪崩: 是指在某一个时间段, 缓存集中过期失效。解决方案, 根据不同情况, 缓存不同周期。用锁/分布式锁或者队列串行访问。
事前:尽量保证整个**redis**集群的高可用性, 发现机器宕机尽快补上。选择合适的内存淘汰策略。
事中: 本地**ehcache**缓存+**hystrix**限流&降级, 避免**Mysql**崩掉
事后: 利用**redis**持久化机制保存的数据尽快回复缓存
- 缓存击穿: 是指一个**key**非常热点, 大并发集中对这个点进行访问, 当这个**key**在失效的瞬间, 持续的大并发就穿破缓存, 直接请求数据库。解决方案, 使用锁, 单机用**synchronized,lock**等, 分布式用分布式锁。缓存过期时间不设置, 而是设置在**key**对应的**value**里。如果检测到存的时间超过过期时间则异步更新缓存。在**value**设置一个比过期时间**t0**小的过期时间值**t1**, 当**t1**过期的时候, 延长**t1**并做更新缓存操作。

1.6 如何保证缓存与数据库双写时的数据一致性

读的时候, 先读缓存, 缓存没有的话, 在读数据库, 然后取出数据放入缓存, 同时返回响应。

更新的时候, 先删除缓存, 在更新数据库。

数据库与缓存更新与读取操作进行异步串行化:

- ①更新数据的时候, 根据数据的唯一标识, 将操作路由之后, 发送到一个**jvm**内部的队列里面去。
- ②读取数据的时候, 如果发现缓存中没有, 那么将从数据库读取数据的操作和更新缓存的操作一起路由到同一个**JVM**内部的队列中去。
- ③一个队列对应一个工作线程, 然后线程从队列里面去取请求进行操作。

LINUX+DOCKER

附录

idea快捷键大全

常用快捷键

- **Ctrl+Shift + Enter**, 语句完成
- **"!"**, 否定完成, 输入表达式时按 **"!"** 键
- **Ctrl+E**, 最近的文件
- **Ctrl+Shift+E**, 最近更改的文件
- **Shift+Click**, 可以关闭文件
- **Ctrl+[OR]**, 可以跑到大括号的开头与结尾
- **Ctrl+F12**, 可以显示当前文件的结构
- **Ctrl+F7**, 可以查询当前元素在当前文件中的引用, 然后按 **F3** 可以选择
- **Ctrl+N**, 可以快速打开类
- **Ctrl+Shift+N**, 可以快速打开文件
- **Alt+Q**, 可以看到当前方法的声明
- **Ctrl+P**, 可以显示参数信息
- **Ctrl+Shift+Insert**, 可以选择剪贴板内容并插入
- **Alt+Insert**, 可以生成构造器/Getter/Setter等
- **Ctrl+Alt+V**, 可以引入变量。例如: `new String();` 自动导入变量定义
- **Ctrl+Alt+T**, 可以把代码包在一个块内, 例如: `try/catch`
- **Ctrl+Enter**, 导入包, 自动修正
- **Ctrl+Alt+L**, 格式化代码
- **Ctrl+Alt+I**, 将选中的代码进行自动缩进编排, 这个功能在编辑 **JSP** 文件时也可以工作
- **Ctrl+Alt+O**, 优化导入的类和包
- **Ctrl+R**, 替换文本
- **Ctrl+F**, 查找文本
- **Ctrl+Shift+Space**, 自动补全代码
- **Ctrl+空格**, 代码提示 (与系统输入法快捷键冲突)
- **Ctrl+Shift+Alt+N**, 查找类中的方法或变量
- **Alt+Shift+C**, 最近的更改
- **Alt+Shift+Up/Down**, 上/下移一行
- **Shift+F6**, 重构 – 重命名
- **Ctrl+X**, 删除行
- **Ctrl+D**, 复制行
- **Ctrl+/或Ctrl+Shift+/, 注释 (//或者/**/)**
- **Ctrl+J**, 自动代码 (例如: `serr`)
- **Ctrl+Alt+J**, 用动态模板环绕
- **Ctrl+H**, 显示类结构图 (类的继承层次)
- **Ctrl+Q**, 显示注释文档
- **Alt+F1**, 查找代码所在位置
- **Alt+1**, 快速打开或隐藏工程面板
- **Ctrl+Alt+left/right**, 返回至上次浏览的位置
- **Alt+left/right**, 切换代码视图
- **Alt+Up/Down**, 在方法间快速移动定位
- **Ctrl+Shift+Up/Down**, 向上/下移动语句
- **F2 或 Shift+F2**, 高亮错误或警告快速定位
- **Tab**, 代码标签输入完成后, 按 **Tab**, 生成代码
- **Ctrl+Shift+F7**, 高亮显示所有该文本, 按 **Esc** 高亮消失
- **Alt+F3**, 逐个往下查找相同文本, 并高亮显示
- **Ctrl+Up/Down**, 光标中转到第一行或最后一行下
- **Ctrl+B/Ctrl+Click**, 快速打开光标处的类或方法 (跳转到定义处)
- **Ctrl+Alt+B**, 跳转到方法实现处
- **Ctrl+Shift+Backspace**, 跳转到上次编辑的地方
- **Ctrl+O**, 重写方法
- **Ctrl+Alt+Space**, 类名自动完成
- **Ctrl+Alt+Up/Down**, 快速跳转搜索结果
- **Ctrl+Shift+J**, 整合两行
- **Alt+F8**, 计算变量值
- **Ctrl+Shift+V**, 可以将最近使用的剪贴板内容选择插入到文本
- **Ctrl+Alt+Shift+V**, 简单粘贴
- **Shift+Esc**, 不仅可以把焦点移到编辑器上, 而且还可以隐藏当前 (或最后活动的) 工具窗口
- **F12**, 把焦点从编辑器移到最近使用的工具窗口
- **Shift+F1**, 要打开编辑器光标字符处使用的类或者方法 **Java** 文档的浏览器
- **Ctrl+W**, 可以选择单词继而语句继而行继而函数
- **Ctrl+Shift+W**, 取消选择光标所在词
- **Alt+F7**, 查找整个工程中使用地某一个类、方法或者变量的位置
- **Ctrl+I**, 实现方法
- **Ctrl+Shift+U**, 大小写转化
- **Ctrl+Y**, 删除当前行
- **Shift+Enter**, 向下插入新行
- **psvm/sout, main/System.out.println(); Ctrl+J**, 查看更多

- Ctrl+Shift+F, 全局查找
- Ctrl+F, 查找/Shift+F3, 向上查找/F3, 向下查找
- Ctrl+Shift+S, 高级搜索
- Ctrl+U, 转到父类
- Ctrl+Alt+S, 打开设置对话框
- Alt+Shift+Inert, 开启/关闭列选择模式
- Ctrl+Alt+Shift+S, 打开当前项目/模块属性
- Ctrl+G, 定位行
- Alt+Home, 跳转到导航栏
- Ctrl+Enter, 上插一行
- Ctrl+Backspace, 按单词删除
- Ctrl+"+/-", 当前方法展开、折叠
- Ctrl+Shift+"+/-", 全部展开、折叠

【调试部分、编译】

- Alt+Shift+F9, 选择 Debug
- Ctrl+F2, 停止
- Alt+Shift+F10, 选择 Run
- Ctrl+Shift+F9, 编译
- Ctrl+Shift+F10, 运行
- Ctrl+Shift+F8, 查看断点
- F8, 步过
- F7, 步入
- Shift+F7, 智能步入
- Shift+F8, 步出
- Alt+Shift+F8, 强制步过
- Alt+Shift+F7, 强制步入
- Alt+F9, 运行至光标处
- Ctrl+Alt+F9, 强制运行至光标处
- F9, 恢复程序
- Alt+F10, 定位到断点
- Ctrl+F8, 切换行断点
- Ctrl+F9, 生成项目
- Alt+1, 项目
- Alt+2, 收藏
- Alt+6, TODO
- Alt+7, 结构
- Ctrl+Shift+C, 复制路径
- Ctrl+Alt+Shift+C, 复制引用, 必须选择类名
- Ctrl+Alt+Y, 同步
- Ctrl+~, 快速切换方案(界面外观、代码风格、快捷键映射等菜单)
- Shift+F12, 还原默认布局
- Ctrl+Shift+F12, 隐藏/恢复所有窗口
- Ctrl+F4, 关闭
- Ctrl+Shift+F4, 关闭活动选项卡
- Ctrl+Tab, 转到下一个拆分器
- Ctrl+Shift+Tab, 转到上一个拆分器

【重构】

- Ctrl+Alt+Shift+T, 弹出重构菜单
- Shift+F6, 重命名
- F6, 移动
- F5, 复制
- Alt+Delete, 安全删除

- Ctrl+Alt+N, 内联

【查找】

- Ctrl+R, 替换
- Ctrl+F, 查找
- F3, 查找下一个
- Shift+F3, 查找上一个
- Ctrl+Shift+F, 在路径中查找
- Ctrl+Shift+R, 在路径中替换
- Ctrl+Shift+S, 搜索结构
- Ctrl+Shift+M, 替换结构
- Alt+F7, 查找用法
- Ctrl+Alt+F7, 显示用法
- Ctrl+F7, 在文件中查找用法
- Ctrl+Shift+F7, 在文件中高亮显示用法